```
// Computer Program Listing Appendix Under 37 CFR 1.52(e)
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
Description of source code files and their contents
code_fragments
 - contains representative code fragments and whole functions
   for encrypted columns during from the PARSE, NORMALIZE,
   PREPROCESS, COMPILE and EXECUTE phases of ASE.  Also
   contains relevant fragments of header files.
crtencrkey.c
 - Entire source module of lower level functions for creating
   an encryption key
encols.c
 - Entire source module of functions to decrypt and
   encrypt columns
encryption.c
 - Entire source module of functions that interface between
   Adaptive Server and the Security Builder API
encolsadmin.c
 - Entire source module of functions to support setting
   the system encryption password
encryption.h
encryptkey.h
 - Encryption-related header files
encryption
 - SQL text of stored procedure sp_encryption used to administer
   encrypted columns
// code_fragments
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
PARSE
Fragments of /calm/svr/sql/generic/source/parser/sql.y
-----------------------------------------------------
/*
** CREATE ENCRYPTION KEY
*/
create_encryption_key: _CREATE_ENCRYPTION_KEY
  {
   /* Make sequence, command, and root nodes */
   r = addstep(TNULL, ENCRKEYCREATE, $1.o, $1.l);
  }
  key_name default_option for_algorithm_clause
  {
   TREE    *resdom;
   /* Set rootname to keyname */
   r->sym.root.rootname = $3;
   /* Attach algorithm to left of root */
   resdom = r->left =  $5;
   targlast = $5;
   if ($4.v)
   {
    /* Set default key status bit */
    r->sym.root.root7stat |= R7T_DEFAULT_KEY;
   }
  }
  with_keyoption_list
;
key_name object
  {
   . . .
   $$ = MKVCHAR($1.v);
  }
;
default_option: /* null */
  {
   $$.v = FALSE;
  }
 | _AS _DEFAULT
  {
   $$.v = TRUE;
```

```
   }
;
for_algorithm_clause: _FOR_AES
   {
    $$ = mkresdom(TNULL, mkvarchar("AES", 3), BNULL, 0);
    $$->sym.resdom.resstat5 |= RES5_ENCRSYM_ALGORITHM;
   }
;
with_keyoption_list: /* null */
   {
    int32   key_size;
    /* default size is 128 bits for AES algorithm
    */
    key_size = EN_AES_DEFAULT_BIT_KEYSIZE;
    targlast->left = mkresdom(TNULL, mkint4(&key_size),
       BNULL, 0);
    targlast = targlast->left;
    /* default is to use initialization vector */
    r->sym.root.root7stat |= R7T_INIT_VECTOR;
   }
  | _WITH keyoption
   {
    if (r->left->left->sym.resdom.resstat5 &
     RES5_ENCRYPT_DEFAULT_LEN)
    {
     if (r->left->left->left)
     {
      /*
      ** Make sure that keylength is specified
      ** when keyvalue is specified
      */
      parserr3(PRS3_KEYLEN_NOT_SPECIFIED,
       $1.1, 1,
     }
    }
    else if (!(r->sym.root.root7stat &
      R7T_RANDOM_PAD) && !initvec_pad)
    {
     . . .
     parserr(P_SYNTAXERR, $1.1, 1, 65,
       "with");
    }
   }
keyoption: keysize_option
   {
    targlast->left = $1;
    $$ = targlast = $1;
   }
  passwd_option
{
 if ($3)
 {
  targlast->left = $3;
  $$ = targlast = $3;
 }
}
initvec_opt
{
 if ($5.v)
 {
  r->sym.root.root7stat |= R7T_INIT_VECTOR;
 }
}
randompad_option
{
 if ($7.v)
 {
  r->sym.root.root7stat |= R7T_RANDOM_PAD;
```

```
    }
   }
   keyvalue_option
   {
    if ($11)
    {
     targlast->left = $11;
     $$ = targlast = $11;
    }
   }
;
keysize_option: /* null */
   {
    long option;
    /* Look up option - if not found, error */
    option = (int) optlookup((char *)$1.v.bval,
      (int) $1.v.blen, Keysizeopt);
    if (option < 0)
    {
     parserr(P_OPTION1, $1.l, 17,
      (BYTE *)($1.v.blen), $1.v.bval, PH_OPT);     YYERROR;
    }
    else
    {
     $$ = mkresdom(TNULL, mkint4(&$2.v), BNULL, 0);
    }
   }
;
. . .
    }
/*
** ALTER ENCRYPTION KEY
*/
alter_encryption_key: _ALTER_ENCRYPTION_KEY
   {
    /* Make sequence, command, and root nodes */
    r = addstep(TNULL, ENCRKEYALTER, $1.o, $1.l)
   }
   key_name optional_default_key_clause
   {
    /* Set rootname to keyname */
    r->sym.root.rootname = $3;
   }
;
optional_default_key_clause: optional_as optional_not_default _DEFAULT
   {
    if (!($2.v) && $3.v)
    {
     /* Set default status key bit */
     r->sym.root.root7stat |= R7T_DEFAULT_KEY;
    }
   }
;
optional_not_default: /* null */
   {
    $$.v = FALSE;
   }
  | _NOT
   {
    r->sym.root.root7stat |= R7T_NOT_DEFAULT_KEY;
    $$.v = TRUE;
   }
;
. . .
/*
** DROP ENCRYPTION KEY
*/
drop_encryption_key: _DROP_ENCRYPTION_KEY
```

```
   {
    /* Make sequence, command, and root nodes */
    r = addstep(TNULL, ENCRKEYDROP, $1.o, $1.l);
   }
   key_name
   {
    /* Attach keyname to the root */
    r->left = mkresdom(TNULL, $3, BNULL, 0);
   }
;
. . .
/*
** CREATE TABLE
*/
create_table:   _CREATE crtab_option _TABLE
   {
    . . .
   }
   object  '('  table_elem_list comma_paren
   {
   }
   . . .
;
. . .
table_elem_list:    table_element
   {
    . . .
   }
;
table_element: col_def opt_default opt_identity col_constr_list
     opt_encrypt opt_storage_type
   {
    . . .
    if ($5)
    {
     . . .
     /* Encryption qualifier was used */
     ENCRYPTION_RESDOM_ASSIGN($$->sym.resdom);
     /* Set the encrypted columns bit in the root */
     ENCRYPTION_ROOT_ASSIGN(r->sym.root);
    }
   }
   . . .
.
================================================================
NORMALIZE
Fragments of /calm/svr/sql/generic/source/sequencer/colnames.c
----------------------------------------------------------------
/*
** COL__FILL_RESDOM_INFO()
**
** Utility work-horse routine called from normalization phase for
** non-SELECT DML queries, and for ALTER TABLE commands.
**
** Walk the resdom list, and validate each column against syscolumns
** to see if it exists. If so, fill in the data type information for
** the column. Otherwise, raise error about 'column not found', and
** return.
*/
col__fill_resdom_info(..., SYB_BOOLEAN &has_encrypt, ...)
{
 . . .
 if (col_found)
 {
  /* Fill in data type info in RESDOM from syscolumns */
  do_fillresd(syscol_sdes, resd);
  if (ENCRYPTION_RESDOM_HAS(resd->sym.resdom))
```

```
   {
    *has_encrypt = TRUE;
   }
   . . .
  }
  . . .
}
/*
** COL__FILL_RESDOMS_BY_NAME
**
** The query is such that it references one or more columns by name.
** . . .
*/
col__fill_resdoms_by_name(...)
{
 . . .
 /* Process the RESDOM list and check for errors */
 if (!col__fill_resdom_info(. . . , &encrypt_has, . . .)
 . . .
 if (encrypt_has)
 {
  ENCRYPTION_ROOT_ASSIGN(root->sym.root);
  ENCRYPTION_RANGEP_ASSIGN(resrg);
 }
 . . .
}
/*
** DO_FILLRESD
**
** Populate a single RESDOM.
** . . .
*/
void
do_fillresd(SDES * s, TREE * resd)
{
 . . .
 COLUMN  columnval;
 . . .
 copyrow((int)SYSCOLUMNS, (BYTE *) s->srow, lencol, (BYTE *) &columnval)
;
 . . .
 if (ENCRYPTION_COLUMN_HAS(columnval))
 {
  if (CIPHERTEXT_IS_ON(pss))
  {
   SET_RESDOM_CIPHERTEXT(resd->sym.resdom,
    columnval.cencrtype, columnval.cencrlen);
   pss->pcurseq->sym.seqnode.seqstat2
    |= SEQ2_CIPHERTEXT_ON;
  }
  else
  {
   ENCRYPTION_RESDOM_ASSIGN(resd->sym.resdom);
  }
 }
}
/*
** COL__FILL_RESDOMS_FOR_TABLE
**
** The query is such that it references no columns by name, and wishes
** to access all (or some) columns in the specified table being updated.
**
** Examples are:
**
**  insert into t1 values (1, 2, 3)
**
** . . .
*/
```

```
SYB_STATIC void
col__fill_resdoms_for_table(TREE * root, SDES * syscol, VRANGE *resrg)
{
 . . .
 /* Scan syscolumns */
 while (getnext(syscol))
 {
  . . .
  do_fillresd(syscol, resd);
  if (ENCRYPTION_RESDOM_HAS(resd->sym.resdom))
  {
   ENCRYPTION_ROOT_ASSIGN(root->sym.root);
   ENCRYPTION_RANGEP_ASSIGN(resrg);
  }
  . . .
 }
}
/*
** MAPVARNODES
**
** Map the column name nodes in the given nodelist (built by
** mkcollist).
*/
mapvarnodes(...)
{
 . . .
    if (ENCRYPTION_COLUMN_HAS(columnval))
    {
     if (CIPHERTEXT_IS_ON(pss))
     {
      /*
      ** Treat varnode as
      ** varbinary
      */
      SET_VARNODE_CIPHERTEXT
          (node->sym.var,
          columnval.cencrtype,
          columnval.cencrlen);
      pss->pcurseq->sym.seqnode
          .seqstat2 |=
          SEQ2_CIPHERTEXT_ON;
     }
     else
     {
      /*
      ** Set encryption status
      */
      ENCRYPTION_VAR_ASSIGN
          (node->sym.var);
      ENCRYPTION_ROOT_ASSIGN
          (root->sym.root);
      ENCRYPTION_RANGEP_ASSIGN(rg);
     }
    }
    . . .
}
================================================================================
===
PREPROCESS (extension of NORMALIZE)
Fragment of /calm/svr/sql/generic/include/trees.h
/*
**  RANGE
**  The range table provides information about the use of a
**  specific object in the database (ie anything with a table id).
*/
typedef struct range
{
 . . .
```

```
 /*
 ** The following field is not saved on disk and should be NULL
 ** when read from disk
 */
 struct rgnondiskres *rgnondiskres; /* ptr to the structure containing
        ** all the non-disk resident fields
        */
 . . .
} VRANGE;
/*
** RGNONDISKRES
** This structure houses all the fields which are not disk resident.
*/
typedef struct rgnondiskres
{
 . . .
 RG_ENCR_KEY *rgencrkey; /* List of encryption key elements */
 RG_ENCR_KEY *rglastencrkey; /* Last pointer to encryption key info
     */
}
/*
** RG_ENCR_KEY - Structure to hold key information from sysencryptkeys.
** This structure is filled during preprocessing, and it is not saved
** in the tree written out to sysprocedures.
*/
typedef struct rg_encr_key
{
 struct rg_encr_key *reknext; /* link */
 objid_t   rekid;  /* Key's object id */
 dbid_t   rekdbid; /* Key's db */
 BYTE   rekvalue[EK_MAX_SYMKEY_VALUE_LEN];
      /* Encrypted key and salt */
 int   reklen;  /* Key size */
 BYTE   rekpasswd[EK_ONDISK_VSLTLEN];
      /* version, salt & sentinel */
 int16   rektype; /* Key type */
 int32   rekstatus; /* Key status */
} RG_ENCR_KEY;
Fragments of /calm/svr/sql/generic/source/sequencer/s_preprocess.c
------------------------------------------------------------------
/*
** S_PREPROCESS
**
** This routine takes the normalized query tree and resolves
** views, aggregates, defaults, and select_into in each sequence step.
** Each of these operations will either modify or expand the query tree.
** . . .
*/
s_preprocess(TREE * seq)
{
 . . .
 /*
 ** Traverse the seq tree and preprocess each root.
 */
 do
 {
  . . .
  {
   . . .
   if (pre_aggview(seq, root, &setp, cmdp, &cmd, TRUE)
      == RECOMPILE)
   . . .
  }
 }
}
/*
** PRE_AGGVIEW
**
```

```
** This routine performs common functions i.e. view resolution and
** aggregate processing ...
*/
pre_aggview(. . .)
{
 . . .
 /*
 ** Call routine to add encrypt/decrypt builtin for base tables.
 */
 if (ENCRYPTION_ROOT_HAS(root->sym.root))
 {
  if (!encr_getinfo(seq, root))
  {
   return FALSE;
  }
  . . .
 }
 . . .
}
/*
**  ENCR_GETINFO
**
** This routine opens syscolumns and for each column that is encrypted,
** gets the keyid, dbid, encrypted type and length. It also gets info
** from sysencryptkeys for each referenced key not seen so far and makes
** a RG_ENCR_KEY element.
**
** It modifies the tree to add encrypt builtin above the value to be
** encrypted and the decrypt builtin above the VAR node column in the tr
ee.
** Keyid, dbid and result type and length of COL_ENCRYPT builtin are fro
m
** syscolumns (encrypted types and lengths). For COL_DECRYPT, type and
** length of argument get the encrypted type and length.
** . . .
*/
SYB_BOOLEAN
encr_getinfo(TREE * seq, TREE * root)
{
 short  cmdtype;
 TREE  *resdom;
 cmdtype = root->sym.root.querytype;
 /*
 ** Traverse the tree looking for ENCRYPTION_VAR_HAS for VAR nodes.
 */
 add_decrypt_bi(root, root, (TREE *)NULL);
 /*
 ** Traverse the tree looking for RESDOMs with ENCRYPTION_RESDOM_HAS
 ** for insert/update
 */
 for (resdom = root->left; resdom; resdom = resdom->left)
 {
  if ((cmdtype != SELECT) &&
   (ENCRYPTION_RESDOM_HAS(resdom->sym.resdom)))
  {
   add_encrypt_bi(root, resdom);
   ENCRYPTION_RESDOM_CLEAR(resdom->sym.resdom);
  }
 }
 /*
 ** Turn off the encryption bit in varnode. This is to make sure
 ** that the nodes which have been processed already do not get
 ** processed again when called for var nodes for views.
 */
 encryption_off(root);
 seq->sym.seqnode.seqstat2 |= SEQ2_HAS_ENCRYPTION;
 return TRUE;
}
```

```
/*
** ADD_DECRYPT_BI
**
** This routine is called to add decrypt builtins.
** It traverses the tree looking for VAR nodes with the encryption statu
s
** bit set and adds the decrypt builtin above the VAR node.
**
** It opens syscolumns and for each column that is encrypted,
** gets the keyid, dbid, encrypted type and length. It also gets info
** from sysencryptkeys for each referenced key not seen so far and makes
** a RG_ENCR_KEY element.
**
** It modifies the tree to add decrypt builtin above the VAR node.
** Keyid, dbid and result type and length of COL_DECRYPT
** builtin are from syscolumns.
**
** . . .
*/
SYB_STATIC void
add_decrypt_bi(TREE *root, TREE *node, TREE *parent)
{
 TREE  *varnode;
 int  child;
 colid_t  colid;
 VRANGE  *rg;  /* range entry */
 TREE  *recurse;
 objid_t  objid;  /* object id of table */
 int32  dbid;  /* database id */
 COLUMN  coldes;  /* runtime column row structure */
 int  namelen;
 /*
 ** Check for stack overflow in this recursive routine.
 ** This must appear immediately after the declarations.
 */
 CHECKSTACKOFLOW;
 varnode = (TREE *) NULL;
 /*
 ** Traverse the tree looking for varnodes with ENCRYPTION_VAR_HAS
 */
 while (node)
 {
  varnode = (TREE *) NULL;
  if (VAR_NODE(node) && (ENCRYPTION_VAR_HAS(node->sym.var)))
  {
   rg = ROOTRG(root, node->sym.var.varno);
   if (!(rg->rgstat & RG_VIEW))
   {
    /*
    ** If the range entry belongs to a view,
    ** do not process this VAR node. This will
    ** be processed during view resolution.
    */
    varnode = node;
    if (node == parent->right)
    {
     child = AOP_RIGHTCHILD;
    }
    else
    {
     child = AOP_LEFTCHILD;
    }
   }
 }
 if (varnode)
 {
  colid = varnode->sym.var.colid;
  dbid = rg->rgdbid;
```

```
objid = rg->rgtabid;
/* Call getcolinf() to get syscolumns info */
if (getcolinf(objid, colid, dbid, &coldes, &namelen)
     == 0)
{
 if (!rg->rgnondiskres->rgdblen)
 {
  getdbname(dbid,
   rg->rgnondiskres->rgdbname,
                                  &rg->rgnondiskres->rgdblen);
 }
 ex_raise(BULKINS, BLK_BADSCHEMA, EX_MISSING, 2,
  rg->rgnlen, rg->rgname,
  rg->rgnondiskres->rgdblen,
  rg->rgnondiskres->rgdbname);
 ex_raise(SYSTEM, SYS_XACTABORT, EX_CONTROL, 0);
}
make_decryption_bi(coldes, varnode, child, parent);
add_rg_encrkey(coldes.cencrkeyid, coldes.cencrkeydb,
  rg);
MEMZERO(&coldes, sizeof(COLUMN));
   }
   /* Stop the search at the leaf level */
   if (LEAFNODE(node))
   {
    break;
   }
   /* Recurse on one side */
   recurse = RECURSE(node);
   if (recurse)
   {
    add_decrypt_bi(root, recurse, node);
   }
   /* Loop on the other side */
   parent = node;
   node = LOOP(node);
 }
}
. . .
/*
** ADD_RG_ENCRKEY
**
** This routine is called to add key info to rg_encr_key struct in
** RGNONDISKRES.
**
** . . .
*/
SYB_STATIC void
add_rg_encrkey(objid_t keyid, dbid_t dbid, VRANGE *rg)
{
 LOCALPSS(pss);
 RG_ENCR_KEY *rg_encr;
 PROC_HDR *hdr = pss->phdr;
 SYB_BOOLEAN key_present;
 ENCRYPTKEY encrkey;
 key_present = FALSE;
 /*
 ** Check if key already present
 */
 for (rg_encr = RG_ENCRKEY_INFO(rg); rg_encr; rg_encr = rg_encr->reknext
)
  {
   if (rg_encr->rekid == keyid && rg_encr->rekdbid == dbid)
   {
    /* Key already present in range structure  */
    key_present = TRUE;
    break;
   }
```

```
}
if (!key_present)
{
 /* Open sysencryptkeys and get key info */
 if (!get_encrkeyinfo(dbid, &keyid, FALSE, &encrkey))
 {
  ex_callprint(EX_NUMBER(ENCRYPTION, ENCR_KEY_NOT_PRESENT), EX_MISSING,
1, TOKENSTR(pss->pcurcmd), keyid);
  ex_raise(SYSTEM, SYS_XACTABORT, EX_CONTROL, 0);
 }
 /* Make a RG_ENCR_KEY element */
 rg_encr = mk_rg_encr_key(hdr);
 rg_encr->rekid = keyid;
 rg_encr->rekdbid = dbid;
 MEMMOVE(encrkey.ekvalue, rg_encr->rekvalue,
  EK_MAX_SYMKEY_VALUE_LEN);
 rg_encr->reklen = encrkey.eklen;
 MEMMOVE(encrkey.ekpasswd, rg_encr->rekpasswd,
  ENCR_VERS_SLT_LEN);
 rg_encr->rektype |= encrkey.ektype;
 rg_encr->rekstatus |= encrkey.ekstatus;
  MEMZERO(&encrkey, sizeof(ENCRYPTKEY));
  /* Link the encryption key info to the table */
  if (RG_LAST_ENCRKEY_INFO(rg))
  {
   RG_LAST_ENCRKEY_INFO(rg)->reknext = rg_encr;
  }
  else
  {
   RG_ENCRKEY_INFO(rg) = rg_encr;
  }
  RG_LAST_ENCRKEY_INFO(rg) = rg_encr;
 }
}
================================================================================
COMPILE
Fragments of /calm/svr/sql/generic/source/sequencer/s_compile.c
--------------------------------------------------------------------------------
s_compile_stmt(...)
{
 . . .
 if (seq->sym.seqnode.seqstat2 & SEQ2_HAS_ENCRYPTION)
 {
  /*
  ** Collect the encryption key information from the virtual
  ** range tables and put into the e_stmt.
  */
  s__mk_encrinfo(lcl_estmt, pss->pplan);
 }
}
/*
** S__MK_ENCRINFO
**
** Build a linked list of E_ENCRKEYS structures, one elmenent
** for each unique key required for encryption or decryption
** in a DML statement.  Extract the key information from the
** RG_ENCR_KEY structure found in the virtual range table.
**
** The method is to find RG_ENCR_KEY info off the main e_steps
** of each e_stmt, and insert the information into the
** E_ENCRKEYS list off the e_stmt.
**
** Parameters:
**  estmt - Ptr to the plan's first estmt
**  phdr  - Ptr to the plan's proc_hdr for mem allocation
**
** Side Effects:
```

```
**  Builds the e_stmt->e_encrkeys list
**
** Returns:
**  Nothing.
**
*/
SYB_STATIC void
s__mk_encrinfo(E_STMT *estmt, PROC_HDR *phdr)
{
 E_STEP  *m_estep; /* Main estep */
 RG_ENCR_KEY *rgkey;
 E_ENCRKEYS *ekey;
 int  maxvarct;
 int16  schemact2;
 int  i;
 for (m_estep = estmt->e_estep; m_estep; m_estep = m_estep->e_stnext)
 {
  for (last_subor = m_estep;
   last_subor->e_stsubor != NULL;
   last_subor = last_subor->e_stsubor)
  {
   continue;
  }
 maxvarct = last_subor->e_stvarct;
 for (i = 0; i < maxvarct; i++)
 {
  if (m_estep->e_stvirtrg[i] == NULL)
  {
   continue;
  }
  rgkey =
      m_estep->e_stvirtrg[i]->rgnondiskres->rgencrkey;
  while (rgkey != (RG_ENCR_KEY *)NULL)
  {
   if ((ekey = encr_key_lookup(rgkey->rekid,
     rgkey->rekdbid, estmt))
     != (E_ENCRKEYS *)NULL)
   {
    /* Key already saved in estmt */
    rgkey = rgkey->reknext;
    continue;
   }
   /* Allocate and initialize runtime struct */
   ekey = (E_ENCRKEYS *)memalloc(phdr,
     sizeof(E_ENCRKEYS));
   ekey->e_ekid = rgkey->rekid;
   ekey->e_ekdbid = rgkey->rekdbid;
   MEMMOVE(rgkey->rekvalue,ekey->e_ekencrvalue,
     EK_MAX_SYMKEY_VALUE_LEN);
   MEMZERO(ekey->e_ekrawvalue, EN_AES_KEY_BUFLEN);
   ekey->e_eklen = rgkey->reklen;
   MEMMOVE(rgkey->rekpasswd, ekey->e_ekpasswd,
     EK_ONDISK_VSLTLEN);
   if (getobj_crdate_or_schemact(rgkey->rekid,
    rgkey->rekdbid, NULL, &schemact2))
   {
    ekey->e_ekschemact = schemact2;
   }
   ekey->e_ekstatus = rgkey->rekstatus;
   ekey->e_ekLctx_enc = NULL;
   ekey->e_ekLctx_dec = NULL;
   /* Link to top of list */
   ekey->e_eknext = estmt->e_encrkeys;
   estmt->e_encrkeys = ekey;
   rgkey = rgkey->reknext;
  }
 }
}
```

```
}
================================================================================
EXECUTE
Fragments of /calm/svr/sql/generic/include/exec.h
--------------------------------------------------
/*
** E_STMT
**
** A statement node corresponds to a single sequence step
** in the original sequence tree.
** . . .
*/
typedef struct e_stmt
{
 . . .
 struct e_encrkeys  *e_encrkeys; /* key information for encrypted
     ** columns
     */
}
/*
** E_ENCRKEYS - typedef for a list element for keeping information
** about a key during statement execution.  The list has one element
** for each separate key required to encrypt or decrypt during
** execution of a DML statement.  The structure is initialized
** during compilation.  The e_ekrawvalue field is assigned a value
** at the beginning of execution and cleared at the end.  The
** e_ekLctx_enc and e_ekLctx_dec fields are assigned a value when
** the encryption setup is done.
*/
typedef struct e_encrkeys
{
        struct e_encrkeys       *e_eknext;      /* ptr to next key info
*/
        objid_t                 e_ekid;         /* object id of key */
        dbid_t                  e_ekdbid;       /* db id of key */
        BYTE                    e_ekencrvalue[EK_MAX_SYMKEY_VALUE_LEN];
                                                /* encrypted value of ke
y */
        BYTE                    e_ekrawvalue[EN_AES_KEY_BUFLEN];
                                                /* raw key */
        size_t                  e_eklen;        /* bit length of key */
        int                     e_ekpasswd[ENCR_VERS_SLT_LEN];
                                                /* salt/version of key *
/
        int32                   e_ekstatus;     /* status of key */
        void                    *e_ekLctx_enc;  /* encryption key contex
t */
        void                    *e_ekLctx_dec;  /* decryption key contex
t */
} E_ENCRKEYS;
Fragments of /calm/svr/sql/generic/source/sequencer/s_execute.c
--------------------------------------------------------------
/*
** S_EXECUTE()
**
** This routine processes a compiled plan prepared by compile and
** follows a linked list of instructions which instruct it to
** perform various commands.
** . . .
*/
s_execute(...)
{
 . . .
 for ( ; estmt; estmt = NEXTSTMT(lastestep, savestmt))
 {
  . . .
  /*
  ** Decrypt all the encryption keys that will be used
```

```
   ** to encrypt/decrypt data during execution.
   */
   if (estmt->e_encrkeys != (E_ENCRKEYS *)NULL)
   {
    /* First ensure that encryption is configured */
    if (!(CFG_GETCURVAL(cfgencryptedcols)))
    {
     ex_callprint(EX_NUMBER(
          ENCRYPTION, ENCR_NO_CONFIG), EX_USER, 6,
      TOKENSTR(pss->pcurcmd));
     ex_raise(SYSTEM,SYS_XACTABORT, EX_CONTROL, 0);
    }
    if (!(s_decrypt_keys(estmt->e_encrkeys)))
    {
     /* Error already printed */
     ex_raise(SYSTEM,SYS_XACTABORT, EX_CONTROL, 0);
    }
   }
}
Fragments of /calm/svr/sql/generic/source/qryproc/run.c
-------------------------------------------------------
/*
** RUN
**
** RUN is the DataServer's expression processor.  It evaluates
** arithmetic and boolean expressions, aggregates, computes,
** and builtin functions.
** . . .
*/
run(...)
{
 /*
 ** Following two EVALS are for encrypted columns.  For
 ** each there are four arguments:
 ** 1st arg: encryption key's id
 ** 2nd arg: encryption key's dbid
 ** 3rd arg: constant for data to be encrypted or decrypted
 ** 4th arg: constant for intermediate copying
 */
 ACTION E_COLENCRYPT:
  if (!(*(exp_sp-1))->len)
  {
   pc->e_evconst->len = 0;
   NEXTPC()
  }
  /*
  ** For intermediate copying during encryption,
  ** allocate memory, if needed.
  */
len = EN_ENCR_VARLEN((*(exp_sp-1))->len,
    (*(exp_sp-1))->type);
if ((*(exp_sp))->maxlen < len)
{
 ALLOC_HEAP_MEM((*(exp_sp)), len);
 (*(exp_sp))->len = len;
}
if (col_encrypt(*(int32 *)(*(exp_sp-3))->value,
  *(int32 *)(*(exp_sp-2))->value,
  (CONSTANT *) (*(exp_sp-1)),
  pc->e_evconst,
  (CONSTANT *) (*(exp_sp))) != SUCCEED)
{
 /* Error already given */
 ex_raise(SYSTEM, SYS_XACTABORT, EX_CONTROL, 0);
}
NEXTPC()
      ACTION E_COLDECRYPT:
if (!(*(exp_sp-1))->len)
```

```
{
 pc->e_evconst->len = 0;
 NEXTPC()
}
/*
** For intermediate copying during decryption,
** allocate memory, if needed.
** See E_COLENCRYPT for order and type of arguments.
*/
len = EN_ENCR_VARLEN((*(exp_sp-1))->len,
    (*(exp_sp-1))->type);
if ((*(exp_sp))->len < len)
{
 ALLOC_HEAP_MEM((*(exp_sp)), len);
 (*(exp_sp))->len = len;
}
if (col_decrypt(*(int32 *)(*(exp_sp-3))->value,
  *(int32 *)(*(exp_sp-2))->value,
  (CONSTANT *) (*(exp_sp-1)),
  pc->e_evconst,
  (CONSTANT *) (*(exp_sp))) != SUCCEED)
  {
   /* Error already given */
   ex_raise(SYSTEM, SYS_XACTABORT, EX_CONTROL, 0);
  }
 . . .
}
// encryption.h
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
#ifndef ENCRYPTION_H_
#define ENCRYPTION_H_ 1
/*
** EN_GLOBALCTX
** This structure holds pointers to the Security Builder session context
** information.  The fields, which are private to SB, hold callback
** ASE function pointers as follows:
**  en_sbgctx (global Security Builder context)
**      contains memory management callback funcs
**  en_yieldctx (yield context)
**   contains "yield" callback function
**  en_rngctx (random number generation context)
**   contains random reseed callback function
** An initialized EN_GLOBALCTX structure is part of the API of the
** encryption.c module.
*/
typedef struct en_ctx {
 void *  en_sbgctx;
 void *  en_gyieldctx;
 void *  en_grngctx;
} EN_GLOBALCTX;
/*
** EN_LOCALCTX
** This structure holds pointers to the Security Builder encryption/
** decryption context, including the SB key, params and local
** encryption context for a given key.  The structure and its
** fields are instantiated at the beginning of encryption/
** decryption operations and freed at the end.
*/
typedef struct enlocalctx {
 void *  en_sblkey;
 void *  en_sblparams;
 void *  en_sblctx;
} EN_LOCALCTX;
/* Key encryption modes */
/* ENCRCOLS_RESOLVE: More status's to come */
#define EN_INIT_VECTOR  0x0001
#define EN_RANDOM_PAD  0x0002
#define EN_ENCRYPT  0x0004
```

```
#define EN_DECRYPT  0x0008
#define EN_AES_BLOCKSIZE  16
#define EN_AES_128_BIT_KEYSIZE  128
#define EN_AES_192_BIT_KEYSIZE  192
#define EN_AES_256_BIT_KEYSIZE  256
#define EN_AES_KEY_BUFLEN  32 /* for upto 256 bit key */
#define EN_AES_DEFAULT_BIT_KEYSIZE EN_AES_128_BIT_KEYSIZE
#define EN_SYSTEM_BIT_KEYSIZE EN_AES_128_BIT_KEYSIZE
#define EN_SYSTEM_BYTE_KEYSIZE EN_SYSTEM_BIT_KEYSIZE/BITS_PER_BYTE
#define EN_AES_IV_BYTELEN EN_AES_BLOCKSIZE
#define EN_INTRNL_KPARTS_LEN 64
#define EN_AES_DIGEST_LEN 20 /* Same as SB_SHA1_DIGEST_LEN */
#define EN_MAXPWDLEN  64
/* Used to generate static symmetric key */
#define KEY1_INDEX 3
#define KEY1_OFFSET 57
#define KEY2_INDEX 0
#define KEY2_OFFSET 103
#define IV_INDEX 2
#define IV_OFFSET 141
/*
** EN_AES_KEY_BLOCK_LEN
** Return key buflen in bytes for a given keysize.
**
** Parameters:
** _keysize - Length in bits of key
*/
#define EN_AES_KEY_BLOCK_LEN(_keysize)      \
 ((((_keysize/BITS_PER_BYTE) - 1)/EN_AES_BLOCKSIZE) + 1)  \
  * EN_AES_BLOCKSIZE)
/*
** EN_AES_KEY_VALIDLEN
** Returns TRUE of length is one of 128, 192 0r 256, FALSE otherwise.
**
** Parameters:
** _keylength - Length in bits of key
*/
#define EN_AES_KEY_VALIDLEN(keylength)      \
 (keylength == EN_AES_128_BIT_KEYSIZE ||     \
  keylength == EN_AES_192_BIT_KEYSIZE ||     \
  keylength == EN_AES_256_BIT_KEYSIZE)
/*
** GEN_STATIC_ENCR_KEY
** Cover for function that mixes up a static key from multiple
** sources.
**
** Parameters:
** _ctx - global AES context
** _k1, _k2, _k3  - bytes for key
** _k1l, _k2l, _k3l - len of above
** _kres - result buf
** _resl - length of result buf
*/
#define GEN_STATIC_ENCR_KEY(_ctx, _k1, _k1l, _k2, _k2l, _k3, _k3l, \
    _kres, _resl)    \
 en_concatbytes(_ctx, _k1, _k1l, _k2, _k2l, _k3, _k3l,  \
   _kres, _resl)
/* Function prototypes */
/* ENCRCOLS_RESOLVE: API to be extended */
int en_init PROTO((EN_GLOBALCTX *, char **));
void en_cleanup PROTO((EN_GLOBALCTX *));
int en_sha1_digest PROTO((EN_GLOBALCTX *, BYTE *, size_t, BYTE *, size_t
 *,
  char **));
int en_aes_createsymkey PROTO((EN_GLOBALCTX *, size_t, int, BYTE *, size
_t *,
  char **));
int en_aes_beginCryptOper PROTO((EN_GLOBALCTX *, EN_LOCALCTX *,
```

```
  unsigned char *, size_t, int, unsigned char *, int, char **));
int en_aes_encrypt PROTO((EN_GLOBALCTX *, EN_LOCALCTX *, unsigned char *
,
  size_t, unsigned char *, char **));
int en_aes_decrypt PROTO((EN_GLOBALCTX *, EN_LOCALCTX *, unsigned char *
,
  size_t, unsigned char *, char **));
int en_aes_endCryptOper PROTO((EN_GLOBALCTX *, EN_LOCALCTX *, char **));
int en_generateRandomData PROTO((EN_GLOBALCTX *, unsigned char *, size_t
));
int en_concatbytes PROTO((EN_GLOBALCTX *, BYTE *, int, BYTE *, int,
  BYTE *, int, BYTE *, size_t *));
#endif /* ENCRYPTION_H_ */
// encryptkey.h
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
/*
**
**    ENCRYPTKEY.H
**
**      This is the row from Sysencryptkeys table.
**
**    Requires:
**              server.h
**
**
*/
#ifndef ENCRYPTKEY_H_
#define ENCRYPTKEY_H_
#include <encryption.h>
#include <rvm.h>
#define EK_PASSWD_HASH_LEN 20
#define EK_PUBLIC_ENCR_LEN 128
#define EK_ENCRYPT_VALUE_LEN 256
# ifdef NOMEMBER_ALIGNMENT
# pragma PRAGMA_NOMEMBER_ALIGNMENT
# endif /* NOMEMBER_ALIGNMENT */
typedef struct encryptkey
{
 /* row locked table format */
 uint16  erno;  /* row format: row number */
 uint16  estat;  /* row format : status field */
 uint16  evarcols; /* row format: # of var len fields */
 /* B1 stuff */
 B1MBDEF(int16, b1padcol1) /* B1 only: pad for alignment */
 B1MBDEF(SLID, b1senscol) /* B1 only: sensitivity label */
 B1MBDEF(SLID, b1infocol) /* B1 only: information label */
 B1MBDEF(int16, b1padcol2) /* B1 only: pad for alignment */
 /* SYSENCRYPTKEYS fixed len fields */
 int16  ektype;  /* Type of encryption key */
objid_t  encrkeyid; /* object id of encryption key */
int32  ekalgorithm; /* Encryption algorithm associated
    ** with key.
    */
int32  ekstatus; /* Status field */
int16  eklen;  /* user specified length of key */
int16  elen;  /* Length of row */
/* SYSENCRYPTKEYS var len fields */
/* ENCR_RESOLVE: EK_ENCRYPT_VALUE may have to be increased as we
** are not sure of the representation for storing the private key,
** which means we do not know how it will look when its encrypted.
** If we save private keys in ASN format and encrypt that, we'll
** probably need even more space. We might need two value columns
** -- one for symmetric (32 bytes) and one for the private key
** which for most rows will be 0 len.
*/
BYTE  ekvalue[EK_ENCRYPT_VALUE_LEN]; /* Encrypted value of
      ** key
      */
```

```
    suid_t  ekuid;  /* uid of user for login access
         */
    /* ENCRCOLS_RESOLVE: This field is mis-named and mis-sized */
    BYTE  ekpasswd[EK_PASSWD_HASH_LEN]; /* Contains 2 bytes
          ** version, 8 bytes
          ** salt plus sentinel
          */
    objid_t  ekpairid; /* object id of public key used for
         ** key encryption
         */
    /* ENCR_RESOLVE: The array length may have to change. */
    BYTE  ekpublic[EK_PUBLIC_ENCR_LEN];  /* Value of key
          ** encrypted with
                ** public key.
          ** For the key
          ** defining row of an
          ** asymmetric key,
          ** this field contains
          ** the public key.
          */
} ENCRYPTKEY;
# ifdef NOMEMBER_ALIGNMENT
# pragma PRAGMA_MEMBER_ALIGNMENT
# endif /* NOMEMBER_ALIGNMENT */
/* Status bit definitions for ektype */
#define EK_SYMMETRIC 0x1
#define EK_ASYMMETRIC 0x2
#define EK_DEFAULT 0x4
/*
** Please update the max value of ektype when you add a new one.
** This value is used in the print routines to print the string for the
** #define value for this field.
*/
#define EKTYPE_MAX EK_DEFAULT
/* Defines for ekalgorithm */
#define EK_AES 0x00000001
#define EK_RSA 0x00000002
/* Defines for ekstatus */
#define EK_INITVECTOR 0x00000001 /* symmetric key uses initialization ve
ctor
    */
#define EK_RANDOMPAD 0x00000002 /* symmetric key uses random padding */
#define EK_KEYRECOVERY 0x00000004 /* symmetric key encrypted for lost pa
sword
       ** protection
       */
#define EK_LOGINACCESS 0x00000008 /* row contains asymmetric encryption
of
       ** symmetric key for login access
       */
#define EK_LOGINPASS 0x00000010 /* asymmetric key whose private key is
       ** encrypted with login password
       */
#define EK_SYSENCRPASS 0x00000020 /* key encrypted with KOK derived from
       ** system encryption password
       */
/*
** This definition describes the largest possible Sysencryptkeys row as
** it appears on disk. Use it to size row I/O buffers.
*/
#define ENCRK_ROW_BUF_SIZE DOL_MAXBUFSIZE(ENCRYPTKEY,ENCR_VARCOL_COUNT)
/*
** Indication that sp_encryption passwords are in hex format
*/
#define EK_HEX_SYSENCRPASSWD 1
#define EK_STATIC_VERS 0
/* Indices for en_ind_tab global table */
#define EK_SPASS_CAT 0 /* Encryption of sys password for catalogs */
```

```
#define EK_SPASS_REP    1 /* Encryption of sys passwd for replication */
#define EK_UKEY  2 /* Encryption of user key */
/*
** ENCR_LOOKUP - Element of a lookup table for indexing into static
** data-gathering memory.  The generated static key always has at
** least one piece of non-static data in the mix, and that is why
** the field names below for the static data are for data elements
** '2' and '3'.
*/
typedef struct encr_lookup {
 int ek2_ind;
 int ek2_off;
 int ek3_ind;
 int ek3_off;
 int ekiv_ind;
 int ekiv_off;
} ENCR_LOOKUP;
/* Number of bytes of salt used for encrypting passwords and keys */
#define ENCR_SALT_LEN  8
/*
** Size of key plus salt after symmetric encryption
*/
#define EK_SYMKEY_ENCR_LEN(_kbitsize)   \
 (((((_kbitsize)/BITS_PER_BYTE +  \
     ENCR_SALT_LEN) -1)/EN_AES_BLOCKSIZE) * EN_AES_BLOCKSIZE) \
     + EN_AES_BLOCKSIZE)
/*
** Max size of symmetric key + salt, rounded up to blocksize for encrypt
ion.
** Sentinel byte included.  The purpose of the sentinel byte (value 1) i
s
** to avoid the risk of trimming trailing zeros in the event the keyvalu
e
** column is used in a SQL statement.
*/
#define EK_MAX_SYMKEY_VALUE_LEN EK_SYMKEY_ENCR_LEN(EN_AES_256_BIT_KEYSIZ
E) +1
/* Number of bytes for internal encryption algorithm version */
#define ENCR_VERSION_LEN 2
/* Size of concatenated version and salt */
#define ENCR_VERS_SLT_LEN ENCR_VERSION_LEN + ENCR_SALT_LEN
/*
** Size of version and salt in sysencryptkeys.ekpasswd.  This length
** includes an extra sentinel byte.  See above comments regarding
** purpose of sentinel.
*/
#define EK_ONDISK_VSLTLEN ENCR_VERS_SLT_LEN +1
/* Number of bytes to store length of encrypted varying length data  */
#define EN_VARLEN_BYTES  2
/*
** Length of plaintext, plus possible EN_VARLEN_BYTES bytes based on
** source datatype, rounded up to block size.
*/
#define EN_ENCR_VARLEN(_len, _type)     \
 ((((_len + (TOK_SI_FIXEDLEN(_type) ? 0 : EN_VARLEN_BYTES) - 1) \
  / EN_AES_BLOCKSIZE) +1) * EN_AES_BLOCKSIZE)
/*
** Length of the ciphertext column, given the length, type of the
** source column and whether an initialization vector is to be
** used.  Include a suffix byte, used so that trailing 0's are not
** trimmed from the varbinary ciphertext column.
*/
#define EN_ONDISK_CIPHERLEN(_len, _type, _useiv)                  \
 (EN_ENCR_VARLEN(_len, _type) +    \
  (_useiv ? EN_AES_BLOCKSIZE : 0) + 1)
/* Macro to test if root has at least one encrypted column */
#define ENCRYPTION_ROOT_HAS(x)  ((x).root7stat & R7T_ENCRYPTED_COL)
/* Set bit to denote that the root has at least one encrypted col. */
```

```
#define ENCRYPTION_ROOT_ASSIGN(x) (x).root7stat |= R7T_ENCRYPTED_COL
/* Clear encryption bit in root */
#define ENCRYPTION_ROOT_CLEAR(x) (x).root7stat &= ~R7T_ENCRYPTED_COL
/* Does estep have encryption bit set? */
#define ENCRYPTION_ESTEPP_HAS(x) ((x)->e_st7stat & R7T_ENCRYPTED_COL)
/* Set it in range entry to denote that there is at least one encrypted
col */
#define ENCRYPTION_RANGEP_ASSIGN(x) (x)->rgstat3 |= RG3_ENCRYPTED_COL
/* Clear encryption bit in range */
#define ENCRYPTION_RANGEP_CLEAR(x) (x)->rgstat3 &= ~RG3_ENCRYPTED_COL
/* Does range entry have encrypted col? */
#define ENCRYPTION_RANGEP_HAS(x) ((x)->rgstat3 & RG3_ENCRYPTED_COL)
/* Is the column encrypted? */
#define ENCRYPTION_COLUMN_HAS(x) ((x).cstatus2 & COL2_ENCRYPTED_COL)
/* Set encryption bit in column */
#define ENCRYPTION_COLUMNP_ASSIGN(x) (x)->cstatus2 |= COL2_ENCRYPTED_COL
/* Set encryption bit in var */
#define ENCRYPTION_VAR_ASSIGN(x) (x).varstat2 |= VAR2_ENCRYPTED_COL
/* Does the VAR have encryption bit set? */
#define ENCRYPTION_VAR_HAS(x)  ((x).varstat2 & VAR2_ENCRYPTED_COL)
/* Clear encryption bit in var node */
#define ENCRYPTION_VAR_CLEAR(x)  (x).varstat2 &= ~VAR2_ENCRYPTED_COL
/* Does the resdom have encryption bit set? */
#define ENCRYPTION_RESDOM_HAS(x) ((x).resstat5 & RES5_ENCRYPTED_COL)
#define ENCRYPTION_RESDOMP_HAS(x) ((x)->resstat5 & RES5_ENCRYPTED_COL)
/* Set encryption bit in resdom */
#define ENCRYPTION_RESDOM_ASSIGN(x) (x).resstat5 |= RES5_ENCRYPTED_COL
/* Clear encryption bit in resdom */
#define ENCRYPTION_RESDOM_CLEAR(x) (x).resstat5 &= ~RES5_ENCRYPTED_COL
/* Set varnode up for reading ciphertext */
#define SET_VARNODE_CIPHERTEXT(_x, _t, _l) \
{       \
 (_x).coltype = _t;    \
 (_x).colen = _l;    \
 (_x).colprec = 0;    \
 (_x).colscale = 0;    \
}
/* Set resdom for writing ciphertext */
#define SET_RESDOM_CIPHERTEXT(_x,_t,_l) SET_VARNODE_CIPHERTEXT(_x,_t,_l)
/* Column encryption related function prototypes */
int  ea_encrypt_syspasswd
   PROTO((int, int, char *, int, char *, int *));
int  ea_decrypt_syspasswd
   PROTO((int, int, char *, int, BYTE *, int *));
int  col_encrypt
   PROTO((objid_t, dbid_t, struct constant *,
    struct constant *, struct constant *));
int  col_decrypt
   PROTO((objid_t, dbid_t, struct constant *,
    struct constant *, struct constant *));
void            encr_alterkey    PROTO((struct e_step *));
SYB_BOOLEAN     encr_checkpermission  PROTO((struct rvm_mpcr *,
      struct e_step *,
      char *, int, dbid_t, objid_t));
SYB_BOOLEAN     encr_defaultkey_check  PROTO((struct xdes *, struct sdes
 *,
                                              objid_t, struct e_step *
,
                                              dbid_t, SYB_BOOLEAN *,
                                              SYB_BOOLEAN));
void            encr_crtkey      PROTO((struct e_step *));
SYB_BOOLEAN     encr_make_static_key
   PROTO((int, int, BYTE *, int, BYTE *, int, BYTE *,
     size_t *));
SYB_BOOLEAN encr_get_sys_passwd PROTO((char *, int, objid_t, dbid_t,
      BYTE *, int *));
SYB_BOOLEAN     encr_getinfo            PROTO((struct tree *, struct tre
e *));
```

```
SYB_BOOLEAN encr_decrypt_key_n_salt PROTO((BYTE *, BYTE *, BYTE *, int,
      int, BYTE *));
SYB_BOOLEAN encr_encrypt_key_n_salt PROTO((BYTE *, BYTE *, BYTE *, int,
      int, BYTE *));
# endif /* ENCRYPTKEY_H_ */
// crtencrkey.c
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
# include <port.h>        /* always required */
# include  <syb_std.h> /* include first */
# include <dtypes.h>      /* server typedefs */
# include <server.h>      /* always required */
# include <stdio.h>
# include <datetime.h>
# include <object.h> /* required by session.h */
# include <session.h> /* required by Pss.h  */
# include <catalog.h> /* required by session.h */
# include <trees.h> /* required by session.h */
# include <lock.h> /* required by lockmgr.h */
# include <lockmgr.h> /* required by Pss.h  */
# include <exception.h> /* required by Pss.h  */
# include <translate.h> /* required by Pss.h  */
# include <loginrec.h> /* required by Pss.h  */
# include <foucvt.h> /* required by Pss.h  */
# include <pss.h>
# include <column.h>
# include <database.h>
# include <dbtable.h>
# include <index.h>
# include <procedure.h>
# include <page.h>
# include <klink.h>
# include <log.h>
# include <resource.h>
# include <tokens.h>
# include <type.h>
# include <user.h>
# include <derror.h>
# include <rvmerr.h>
# include <crerr.h>
# include <udrerr.h>
# include <encrypterr.h>
# include <schemaerr.h>
# include <utils.h>
# include <exec.h>
# include <sem.h>
# include <parsename.h>
# include <bitbyte.h>
# include <trace.h>
# include <create.h>
# include <timestamp.h>
# include <seqerr.h>
# include <accesserr.h>
# include       <intl.h>
# include       <config.h>
# include <cfg_ds.h>
# include <cfg_mgr.h>
# include <textmgr.h>
# include <syb_nls.h>
# include <ddblkio.h>
# include <dstrmio.h>
# include <tokenop.h>
# include <syb_secure.h>
# include <password.h>
# include <rvm_dcl.h> /* RVM function prototypes */
# include <srvroles.h>
# include <roles.h>
# include <udr.h>
# include <encryptkey.h>
```

```
# include <encryption.h>
#if USE_SECURITYBUILDER
# include <src_dcl.h> /* function prototypes */
# include <bt_rowfmt_public.h>
# include <bt_public.h>
# include <sysattr.h>
# include <attribute.h>
# include <attrdef.h>
# include <tod.h>
# include <ha_states.h>
# include <xactmgr_priv.h>
#define NUMLOCKS 3 /* The total number of locks obtained
                        ** by the CREATE ENCRYPTION KEY command
   */
SYB_STATIC SYB_BOOLEAN encr__crt_symkey PROTO((struct xdes *, struct e_s
tep *,
     char *, int, BYTE *, BYTE *, objid_t, dbid_t,
     suid_t));
SYB_STATIC SYB_BOOLEAN encr__crt_symkey_encrypt PROTO((char *, int, int,

       BYTE *, dbid_t,
       SYB_BOOLEAN, BYTE *,
       int, BYTE *));
extern unsigned char *get1buf PROTO((int, int));
extern unsigned char* get2buf   PROTO((int, int));
extern unsigned char* get3buf   PROTO((int, int));
extern ENCR_LOOKUP en_ind_tab[2][3];
/*
** ENCR_CRTKEY
**
** encr_crtkey takes E_STEP as a parameter and executes the command to c
reate
** encryption keys.
**
** This routine will further call routines encr__crt_symkey() and
** encr__crt_asymkey() to create symmetric keys for AES algorithm and
** asymmetric keys for RSA algorithm.
**
** ENCR_RESOLVE: encr__crt_asymkey() will be supported in the next phase
.
**
** estep->e_stresdom points to a RESDOM which contains the algorithm for
 the
** key to be created (AES for symmetric and RSA for asymmetric). The nex
t
** resdom in the list denotes the keylength to be used in bits. The foll
owing
** resdoms if present denote user defined password and keyvalue respecti
vely.
** The valid keylengths for AES are 128, 192 and 256. For RSA, its 512,
1024
** and 2048 bits.
**
** Parameter:
** estep -- an E_STEP
**
** Caller: s_execute()
**
** Return: None
**
** Side Effects:
**  Adding the created key to system table sysencryptkeys
**
**
*/
void
encr_crtkey(E_STEP *estep)
{
```

```
 LOCALPSS(pss);
 TREE    *resdom;
 XDES    *mxdes;
 char    *keyname;
 int    keyname_len;
 objid_t    new_keyid;
 TRANPARAMS(xprm); /* Parameter list for actions */
 LOCKREQUEST       lock_requests[NUMLOCKS];
 LOCKREQUEST       *lock_requestsp[NUMLOCKS];
     /* Declare two arrays:lock_requestsp[],
                                                ** and lock_requests[].
                                                ** lock_requests[] is an array
                                                ** of LOCKREQUEST;
                                                ** lock_requestsp[] is an array
of
                                                ** pointers to LOCKREQUEST, each
                                                ** of which points to the corres
ponding
                                                ** LOCKREQUEST structure of
                                                ** lock_requests[].
                                                **
                                                ** The array lock_requestsp[] is
                                                ** passed to lock_multiple(), to

     ** obtain all our locks at once.
                                                */
BYTE    ciphertext[EK_ENCRYPT_VALUE_LEN];
int    actual_locks;
int    lock_entry;
suid_t    uid;
dbid_t    dbid;
RVM_MPCR   *rvm_mpcr;
PARSETABLE   ptable;
BYTE    vers_salt[EK_ONDISK_VSLTLEN];
     /* from PASSWD opt */
VOLATILE struct
{
 XDES *mxdes;
 XACTPRM *xprm;
 int db_opened1;
 PSS *pss;
} copy;
/* Keep these backout variables in memory */
SYB_NOOPT(copy);
MEMZERO(&copy, sizeof(copy));
copy.pss = pss;
copy.xprm = &xprm;
copy.db_opened1 = FALSE;
/*
 ** Do checking for configuration variable
 */
 if (!Resource->rconfig->cfgencryptedcols)
 {
  ex_callprint(EX_NUMBER(ENCRYPTION, ENCR_NO_CONFIG), EX_USER, 2, TOKENS
TR(estep->e_stquerytype));
  return;
 }
 /* Exception handling and backout section */
 if (ex_handle(EX_ANY, EX_ANY, EX_ANY, ut_handle))
 {
backout:
  if (copy.mxdes)
  {
   /* Abort any transactions in progress */
   copy.mxdes = (XDES *) NULL;
   /*
   ** Populate the parameter list for
   ** xact_rollback() API.
```

```
  */
  XACTPRM_END(*copy.xprm, NULL, 0, copy.pss, XACT_LOCAL);
  xact_rollback(copy.xprm);
 }
 if (copy.db_opened1)
 {
  copy.db_opened1 = FALSE;
  closedb(USEPREV);
 }
 return;
}
/*
** ENCR_RESOLVE: Need to add auditing for CREATE ENCRYPTION KEY.
** Resolution 354538-3 for this and will be done separately.
*/
dbid = -1;
new_keyid = 0;
rvm_mpcr = (RVM_MPCR *) NULL;
MEMZERO(ciphertext, EK_ENCRYPT_VALUE_LEN);
MEMZERO(vers_salt, EK_ONDISK_VSLTLEN);
resdom = estep->e_stresdom;
keyname = (char *) estep->e_stname->value;
keyname_len = estep->e_stname->len;
/*
** RVM - If permission checking is to be done, set the rvm_mpcr
** which will be used later on to register accesses.
** Fetch the mpcr
*/
SYB_ASSERT(pss->pcurstmt != (E_STMT *) NULL);
       rvm_mpcr = pss->pcurstmt->e_mpcr;
/* Establish database and owner context for key */
if (!(uid = obj_context(&ptable, &keyname, &keyname_len, rvm_mpcr)))
{
 goto backout;
}
copy.db_opened1 = TRUE;
dbid = pss->pcurdb;
/*
** Begin transaction
*/
XACTPRM_LOCAL(xprm, "$createncrkey", 13, NULL, pss->pdbtable,
  BEGINXACT_UPDATE | BEGINXACT_DDL );
if (xact_begin(&xprm) != XACTRV_SUCCESS)
{
 goto backout;
}
copy.mxdes = mxdes = xprm.xdes;
/* get available key id to use */
if (!(new_keyid = obj_id(mxdes)))
{
 goto backout;
}
/*
** Initialize the ENCRKEYCREATE IPCR and attributes and fill in
** the required information. Tag the attribute to the IPCR and
** register the access RVM_ENCRKEYCREATE to do the necessary
** permission checking.
*/
/*
** Is user allowed to perform this command?
*/
if (!encr_checkpermission(rvm_mpcr, estep, keyname, keyname_len,
    dbid, new_keyid))
{
 goto backout;
}
/*
** Get all the locks needed by this CREATE ENCRYPTION KEY command.
```

```
** The LOCKREQUEST structure is defined in lock.h.
        **
        ** When changing the number of locks in this list, always
        ** remember to redefine NUMLOCKS to be equal to the
        ** size of the list.
        */
        actual_locks = 0;
LOCKREQ_ARY_SETUP(lock_requestsp, lock_requests, NUMLOCKS);
        LOCKREQ_INIT(
          lock_requests[actual_locks], EX_TAB, SYSENCRYPTKEYS,
          dbid, LOCKSUFFCLASS_XACT, PCUR_XACTLOCKS(pss),
                LCTX_XACT, actual_locks, NUMLOCKS, 123);
        actual_locks++;
        LOCKREQ_INIT(
          lock_requests[actual_locks], EX_TAB, SYSOBJECTS,
          dbid, LOCKSUFFCLASS_XACT, PCUR_XACTLOCKS(pss),
                LCTX_XACT, actual_locks, NUMLOCKS, 124);
        actual_locks++;
/*
** Get a lock on the object being created, to avoid a sanity check
** if the transaction rolls back.
*/
lock_entry = lock_find_entry(lock_requestsp, NUMLOCKS, actual_locks,
   new_keyid, dbid, LOCK_ALLOC_ALWAYS);
if (lock_entry < 0)
{
 ex_raise(ENCRYPTION, ENCR_LOCK_OVERFLOW, EX_CMDFATAL, 1);
}
        LOCKREQ_INIT(
          (*lock_requestsp[lock_entry]), EX_TAB, new_keyid,
          dbid, LOCKSUFFCLASS_XACT, PCUR_XACTLOCKS(pss),
                LCTX_XACT, actual_locks, NUMLOCKS, 125);
        actual_locks++;
if (lock_multiple(lock_requestsp, actual_locks) < 0)
{
                ex_raise(UTILS, EX_ANY, EX_CONTROL, 0);
        }
/* check if we need to create a symmetric key or asymmetric key */
if (resdom->sym.resdom.resstat5 & RES5_ENCRSYM_ALGORITHM)
{
 if (!encr__crt_symkey(mxdes, estep, keyname, keyname_len,
    ciphertext, vers_salt, new_keyid, dbid, uid))
 {
  goto backout;
 }
}
/* ENCR_RESOLVE: The else part will call encr__crt_asymkey()
** once it is supported
*/
/*
** Log the CREATE ENCRYPTION KEY command if we have DDL replication set
** and we are not called from a stored procedure.
*/
if (DDL_REPLICATION_IS_ON(pss, pss->pdbtable) &&
    !CALLED_FROM_SPROC(pss))
{
 if (!log_encrkey(copy.mxdes, estep,
   (BYTE *) keyname, keyname_len, new_keyid,
   (BYTE *) ciphertext,
   (BYTE *) vers_salt,
   CREATE_ENCR_KEY, LOGCMD_NULL))
 {
   ex_raise(SCHEMA, SCHEMAREP_LOGFAIL, EX_USER, 11,
    strlen(TOKENSTR(estep->e_stquerytype)),
    TOKENSTR(estep->e_stquerytype),
    keyname_len, keyname, pss->pcurdb,
    strlen(TOKENSTR(estep->e_stquerytype)),
    TOKENSTR(estep->e_stquerytype));
```

```
  }
 }
 MEMZERO(ciphertext, EK_ENCRYPT_VALUE_LEN);
 MEMZERO(vers_salt, EK_ONDISK_VSLTLEN);
 xact_commit(&xprm);
 copy.mxdes = (XDES *) NULL;
 mxdes = (XDES *) NULL;
 /* switch back to previous db as current */
 copy.db_opened1 = FALSE;
 closedb(USEPREV);
}
/*
** ENCR__CRTSYMKEY
**
** encr__crt_symkey takes XDES, keyname, keyid, dbid and uid as paramete
rs and
** executes the command to create symmetric encryption keys.
** This routine calls the low level API of Security Builder to actually
create
** the key.
**
** The symmetric key is itself encrypted using the system default passwo
rd if
** no user defined password is specified.
**
** The encrypted key is inserted into sysencryptkeys and the keyname, ui
d
** and the creation date into sysobjects.
**
** Parameters:
**  xdes  -- pointer to XDES
**  e_step  -- pointer to e_step
**  keyname  -- name of key to be created
**  keyname_len -- length of keyname to be created
**  ciphertext -- hexadecimal representation of the key
**  vers_salt -- version and salt
**  keyid  -- keyid of the encryption key
**  dbid  -- dbid of the database where the key is to
**     be created
**  uid  -- uid of the user for whom the key needs to
**     to be created
**
** Caller:  encr_crtkey()
**
** Return: FALSE if key was not created
**  TRUE otherwise
**
** Side Effects:
** Adding the created symmetric key in system table sysobjects and
** sysencryptkeys.
**
*/
SYB_STATIC SYB_BOOLEAN
encr__crt_symkey(XDES *xdes, struct e_step *estep, char *keyname,
  int keyname_len, BYTE *ciphertext, BYTE *vers_salt,
  objid_t keyid, dbid_t dbid, suid_t uid)
{
 LOCALPSS(pss);
 TREE   *temp_resdom;
 TREE   *resdom;
 BYTE   *algorithm;
 int   keylength;
 int   algo_len;
 ENCRYPTKEY  encryptkey;
 int   lencol[ENCR_VARCOL_COUNT];
 int   i;
 BYTE   rowbuf[ENCRK_ROW_BUF_SIZE];
 int   rowlen;
```

```
OBJ_ROWINFO             objnr;          /* for call to obj_newrow() */
int32             objstat2;       /* status2 for sysobjects */
objid_t   targetid;
int                     loginamelen;    /* for concrete id of obj */
char   *sysstatus;
SDES   *sencrkeysdes;
int   keyvallen;
SYB_BOOLEAN  default_key;
SYB_BOOLEAN  using_passwd;
int   pwdlen; /* length of password */
BYTE   pwd[EN_MAXPWDLEN]; /* to hold clear password */
short   vers;
BYTE   *saltp;  /* Pointer to salt */
BYTE   raw_key[EN_AES_KEY_BUFLEN];
BYTE   keystatic[EN_AES_DIGEST_LEN]; /* to mix key */
size_t   kstaticlen; /* Length of static key bytes */
char   *missing_opt;
CONSTANT  *const_arg;
VOLATILE struct
{
 SDES *sencrkeysdes;
 } copy;
 /* Keep these backout variables in memory */
 SYB_NOOPT(copy);
 MEMZERO(&copy, sizeof(copy));
 objstat2 = 0;
 targetid = 0;
 sysstatus = "EK";
 default_key = FALSE;
 using_passwd = FALSE;
 keyvallen = 0;
 pwdlen = 0;
 kstaticlen = EN_AES_DIGEST_LEN;
 vers = 0;
 MEMZERO(&objnr, sizeof(objnr));
 MEMZERO(&encryptkey, sizeof(ENCRYPTKEY));
 resdom = estep->e_stresdom;
 algorithm = resdom->right->sym.constant.value;
 algo_len = resdom->right->sym.constant.len;
 keylength = (*(int *)resdom->left->right->sym.constant.value);
 if (ex_handle(EX_ANY, EX_ANY, EX_ANY, ut_handle))
 {
backout:
  CLOSE_SDES(&copy.sencrkeysdes);
  return (FALSE);
 }
 /*
 ** Check the lengths specified is one of the following: 128, 192,
 ** 256 for AES algorithm. If not, give error.
 */
 if (!EN_AES_KEY_VALIDLEN(keylength))
 {
  ex_raise(ENCRYPTION, ENCR_WRONGLEN, EX_USER, 1, algo_len,
    algorithm, 3, EN_AES_128_BIT_KEYSIZE,
    3, EN_AES_192_BIT_KEYSIZE,
    3, EN_AES_256_BIT_KEYSIZE);
 }
 for (temp_resdom = resdom->left; temp_resdom;
  temp_resdom = temp_resdom->left)
 {
  const_arg = &temp_resdom->right->sym.constant;
  switch (temp_resdom->sym.resdom.resstat5)
  {
   case RES5_ENCRYPT_KEYVALUE:
    MEMMOVE(const_arg->value, ciphertext,
     MIN(EK_ENCRYPT_VALUE_LEN,
     const_arg->len));
    keyvallen = const_arg->len;
```

```
    break;
   case RES5_ENCRYPT_KEYSTATUS:
    encryptkey.ekstatus =
     *(int32 *)const_arg->value;
    break;
   case RES5_ENCRYPT_KEYPASSWD:
    /*
    ** Could be user password (when implemented)
    ** or vers/salt.  We'll know after getting
    ** keystatus.
    */
    pwdlen = const_arg->len;
    MEMMOVE(const_arg->value, pwd,
     MIN(EN_MAXPWDLEN, pwdlen));
    using_passwd =  TRUE;
    break;
   default:
    break;
  }
}
if (keyvallen && using_passwd &&
 (encryptkey.ekstatus & EK_SYSENCRPASS))
{
 if (pwdlen != EK_ONDISK_VSLTLEN)
 {
  ex_raise(ENCRYPTION, ENCR_OPT_LEN, EX_USER, 1,
   "PASSWD", TOKENSTR(estep->e_stquerytype),
   EK_ONDISK_VSLTLEN);
 }
 else
 {
  /* We now know that the passwd is really vers/salt */
  MEMMOVE(pwd, vers_salt, EK_ONDISK_VSLTLEN);
  pwdlen = 0;
 }
 /* KEYVALUE should include sentinel byte */
 if (keyvallen != (EK_SYMKEY_ENCR_LEN(keylength)+1))
 {
  ex_raise(ENCRYPTION, ENCR_OPT_LEN, EX_USER, 1,
   "KEYVALUE", TOKENSTR(estep->e_stquerytype),
   (EK_SYMKEY_ENCR_LEN(keylength)+1));
 }
}
else if (keyvallen || (encryptkey.ekstatus & EK_SYSENCRPASS))
{
 missing_opt = (keyvallen==0) ? "KEYVALUE" : (!using_passwd ?
  "PASSWD" : "KEYSTATUS");
 ex_raise(ENCRYPTION, ENCR_MISSING_OPTION, EX_USER, 1,
  missing_opt, TOKENSTR(estep->e_stquerytype));
}
else if (using_passwd)
{
 /*
 ** Currently use of passwd w/out keyvalue and keystatus
 ** is disallowed.
 */
 ex_raise(ENCRYPTION, ENCR_MISSING_OPTION, EX_USER, 1,
  "KEYVALUE", TOKENSTR(estep->e_stquerytype));
}
if ((!using_passwd) || (keyvallen &&
  (encryptkey.ekstatus & EK_SYSENCRPASS)))
{
 /*
 ** Call routine to get system password from sysattributes
 ** and decrypt it.
 */
 pwdlen = EN_MAXPWDLEN;
 if (!encr_get_sys_passwd(keyname, keyname_len, 0, dbid,
```

```
      &pwd[0], &pwdlen))
  {
   goto backout;
  }
}
/*
** If the keyvalue has been supplied, verify that the
** key can be decrypted in this server.
*/
if (keyvallen && (encryptkey.ekstatus & EK_SYSENCRPASS))
{
 vers = GETSHORT(vers_salt);
 saltp = (vers_salt + ENCR_VERSION_LEN);
 /*
 ** Make a static key out of the system encryption password
 ** password, the supplied salt and a static ingredient.
 */
 if (!encr_make_static_key(EK_UKEY, vers, (BYTE *)&pwd[0],
   pwdlen, saltp, ENCR_SALT_LEN, &keystatic[0],
   &kstaticlen))
 {
  /* ENCRCOLS_RESOLVE: Give error ? */
  return FAIL;
 }
 if (!encr_decrypt_key_n_salt(&keystatic[0], saltp,
  ciphertext, keylength, EN_AES_KEY_BUFLEN,
  &raw_key[0]))
 {
  /* Error already reported */
  goto backout;
 }
 /*
 ** We've validated the key sent in keyvalue, so
 ** throw away the raw key.  Keyvalue is still in
 ** the ciphertext buffer.
 */
 MEMZERO(&raw_key[0], EN_AES_KEY_BUFLEN);
}
/*
** Call routine to hash system default password, create symmetric
** encryption key and encrypt the column encryption key with the
** message digest.
*/
if (!keyvallen)
{
 if (!encr__crt_symkey_encrypt(keyname, keyname_len,
    keylength, ciphertext, dbid,
    using_passwd, pwd, pwdlen,
    vers_salt))
 {
  MEMZERO(pwd, pwdlen);
  goto backout;
 }
}
MEMZERO(pwd, pwdlen);
/*
** Save the encrypted key, algorithm, keylength, status bits to
** sysencryptkeys
*/
sencrkeysdes = OPEN_SYSTAB_BY_DBID(SYSENCRYPTKEYS, dbid);
if (sencrkeysdes == (SDES *) NULL)
{
 open_fail_error((DBTABLE *)UNUSED, dbid,
   (objid_t) SYSENCRYPTKEYS);
}
copy.sencrkeysdes = sencrkeysdes;
sencrkeysdes->sstat |= (SS_FGLOCK | SS_UPDLOCK | SS_L1LOCK);
/*
```

```
** If a default key already exists for
** the database, give error.
*/
if (estep->e_st7stat & R7T_DEFAULT_KEY)
{
 if (encr_defaultkey_check(xdes, sencrkeysdes, (objid_t)NULL,
    estep, dbid, &default_key, FALSE))
 {
  if (default_key)
  {
   ex_raise(ENCRYPTION, ENCR_DEFAULT_EXISTS,
    EX_USER, 1);
  }
 }
 else
 {
  goto backout;
 }
}
/* Initialize all varying length fields to NULL */
for (i = 0; i < ENCR_VARCOL_COUNT; i++)
{
 lencol[i] = 0;
}
if (xact_beginupdate(xdes, sencrkeysdes, XMOD_DIRECT, 0) !=
    XACTRV_SUCCESS)
{
 ex_raise(ENCRYPTION, EX_ANY, EX_CONTROL, 0);
}
encryptkey.encrkeyid = keyid;
encryptkey.ektype |= EK_SYMMETRIC;
encryptkey.ekalgorithm = EK_AES;
if (estep->e_st7stat & R7T_DEFAULT_KEY)
{
 encryptkey.ektype |= EK_DEFAULT;
}
/*
** The resdom below the algorithm resdom has the keylength
*/
encryptkey.eklen = keylength;
if (estep->e_st7stat & R7T_INIT_VECTOR)
{
 encryptkey.ekstatus |= EK_INITVECTOR;
}
if (estep->e_st7stat & R7T_RANDOM_PAD)
{
 encryptkey.ekstatus |= EK_RANDOMPAD;
}
if (!using_passwd)
{
 encryptkey.ekstatus |= EK_SYSENCRPASS;
}
MEMMOVE(ciphertext, encryptkey.ekvalue, EK_ENCRYPT_VALUE_LEN);
MEMMOVE(vers_salt, encryptkey.ekpasswd, EK_ONDISK_VSLTLEN);
lencol[ENCR_VALUE_VLIDX] = EK_ENCRYPT_VALUE_LEN;
lencol[ENCR_EKPASSWD_VLIDX] = EK_ONDISK_VSLTLEN;
rowlen = fmtrow(SYSENCRYPTKEYS, (BYTE *) &encryptkey, lencol, rowbuf);
if (!insert(sencrkeysdes, rowbuf, rowlen))
{
 ex_raise(UTILS, INSERT_FAIL, EX_CONTROL, 0);
}
if (xact_endupdate(xdes) != XACTRV_SUCCESS)
{
 ex_raise(ENCRYPTION, EX_ANY, EX_CONTROL, 0);
}
/* Close sysencryptkeys */
CLOSE_SDES(&copy.sencrkeysdes);
/*
```

```
 ** Save the keyname, creation date and uid into sysobjects.
 ** Setup obj_rowinfo struct for the call to obj_newrow().
 */
 objnr.obj.objostat.objid = keyid;
 MOVE_FIXED(sysstatus, &objnr.obj.objtype[0], sizeof(objnr.obj.objtype))
;
 objnr.obj.objostat.objsysstat = O_ENCRKEY;
 objnr.obj.objostat.objsysstat2 = objstat2;
 SYB_ASSERT(keyname_len <= sizeof (objnr.obj.objname));
 MEMMOVE(keyname, objnr.obj.objname, keyname_len);
 objnr.objlencol[OBJNAMELEN] = keyname_len;
 objnr.obj.objuid = uid;
 /*
        ** DBO-owned objects do not belong to an individual login.
        ** The DBO's suid is retrieved from the dbtable during
        ** permissions checking at execution time.  But if the creator i
s
        ** aliased to the DBO and for all other cases, save the suid.
        */
        if (!(pss->puid == DBO_UID &&
                pss->psuid == dbt_getdbo(dbid, (DBTABLE *)UNUSED)))
        {
                if (getsusername(pss->psuid, objnr.obj.objloginame,
                                &loginamelen, (char *)NULL, (int *)NULL)
)
                {
                        objnr.objlencol[OBJLOGINLEN] = loginamelen;
                }
        }
 /*
 ** objrealname is needed for errors in obj_newrow().
 */
 SYB_ASSERT(keyname_len <= sizeof (objnr.objrealname));
        MEMMOVE(keyname, objnr.objrealname, keyname_len);
        objnr.objrnamelen = keyname_len;
        objnr.objtarget = targetid;
 /* update sysobjects */
 obj_newrow(xdes, &objnr);
 return TRUE;
}
/*
** ENCR_CHECKPERMISSION
**  Check permissions and database validation. Only SSO can
**  do CREATE/ALTER ENCRYPTION KEY AS [NOT] DEFAULT. Users need to
**  have been granted ENCRKEYCREATE/ENCRKEYALTER permission in the
**  database before they can execute the command.
**
** Parameters:
**  rvm_mpcr - pointer to RVM_MPCR
**  estep - pointer to E_STEP
**  keyname - pointer to keyname
**  keyname_lem - pointer to length of keyname
**  dbid - dbid where key needs to be created
**  keyid - id of key being created/altered
**
** Returns:
**  if there is a problem and returns FALSE.
**
** Side Effects:
**
*/
SYB_BOOLEAN
encr_checkpermission(RVM_MPCR *rvm_mpcr, struct e_step *estep, char *key
name,
   int keyname_len, dbid_t dbid, objid_t keyid)
{
 LOCALPSS(pss);
 RVM_IPCR  rvm_ipcr;
```

```
RVM_DB_ID  attr_dbid;
RVM_SYSOBJ_OBJ_NAME keyname_attr; /* attribute structure to
     ** contain name of the
     ** encryption key
     */
RVM_SYSOBJ_OBJ_ID       attr_objid;
RVM_OPTION    option_attr; /* option attribute */
RVM_TREE_ATTR  tree_attr;
int               curcmd;
       SYB_ASSERT(rvm_mpcr != (RVM_MPCR *) NULL);
       curcmd   = RVM_COMMAND(rvm_mpcr);
       SYB_ASSERT(curcmd == (int) pss->pcurcmd);
/*
** Initialize the IPCR
*/
if (rvm_ipcr_init(SI_XLATE_AC(curcmd), &rvm_ipcr) == RVM_ERROR)
{
 ex_callprint(EX_NUMBER(RVM, RVM_INTERNAL_ERR), EX_INTOK, 295);
 return FALSE;
}
if (estep->e_stquerytype == ENCRKEYCREATE)
{
 /* Fill in the dbid where the key is being created */
  if (rvm_init_attr(RVM_ATTR_OBJ_ACCESSED, RVM_INFO_DB_ID,
   &attr_dbid, sizeof(attr_dbid)) == RVM_ERROR)
 {
  ex_callprint(EX_NUMBER(RVM, RVM_INTERNAL_ERR), EX_INTOK, 296);
  return FALSE;
 }
 /* Fill-in the dbid where the key is being created */
 attr_dbid.dbid = dbid;
 /* attach the attribute for db id */
 if (rvm_attach_attr((ATTR_HDR *)&attr_dbid, &rvm_ipcr, rvm_mpcr)
   == RVM_ERROR)
 {
  ex_callprint(EX_NUMBER(RVM, RVM_INTERNAL_ERR), EX_INTOK, 297);
  return FALSE;
 }
 /* Initialize the attribute for the encryption key name */
 if (rvm_init_attr(RVM_ATTR_NAME, RVM_INFO_SYSOBJ_OBJ_NM,
  &keyname_attr, sizeof(keyname_attr)) != RVM_OK)
 {
  ex_callprint(EX_NUMBER(RVM, RVM_INTERNAL_ERR), EX_INTOK, 298);
  return FALSE;
 }
 /* fill in the attribute information */
 MEMMOVE(keyname, keyname_attr.obj_name.name, keyname_len);
 keyname_attr.obj_name.len = keyname_len;
 /* Attach the name attribute to the ipcr */
   if (rvm_attach_attr((ATTR_HDR *)&keyname_attr, &rvm_ipcr,
     rvm_mpcr) != RVM_OK)
   {
    ex_callprint(EX_NUMBER(RVM, RVM_INTERNAL_ERR), EX_INTOK, 299);
    return FALSE;
   }
  }
  else
  {
   /*
   ** This is for ENCRKEYALTER.
   */
   if (rvm_init_attr(RVM_ATTR_OBJ_ACCESSED, RVM_INFO_SYSOBJ_OBJ_ID,
                          &attr_objid, sizeof(attr_objid)) != RVM_OK)
                  {
                          ex_callprint(EX_NUMBER(RVM, RVM_INTERNAL_ERR), E
X_INTOK,
 303);
    return FALSE;
```

```
                }
                /* Fill-in the attribute information */
                attr_objid.obj_id = keyid;
                attr_objid.db_id  = dbid;
                /* Attach the attribute and register the access. */
  if (rvm_attach_attr((ATTR_HDR *)&attr_objid, &rvm_ipcr,
   rvm_mpcr) != RVM_OK)
                {
                        ex_callprint(EX_NUMBER(RVM, RVM_INTERNAL_ERR), E
X_INTOK,
 304);
   return FALSE;
                }
 }
 /* Initialize the attribute which will contain the command's option. */
        if (rvm_init_attr(RVM_ATTR_ENCR_OPTION, RVM_INFO_OPTION,
                        (void *) &option_attr, sizeof(option_attr))
                != RVM_OK)
        {
                ex_callprint(EX_NUMBER(RVM, RVM_INTERNAL_ERR), EX_INTOK,
 300);
 return FALSE;
        }
        /*
        ** Get the option in the command and fill-in
        ** the attribute information.
        */
if (estep->e_st7stat & (R7T_DEFAULT_KEY | R7T_NOT_DEFAULT_KEY))
{
 option_attr.option |= estep->e_st7stat;
}
/* Attach the option command attribute to the IPCR. */
        if (rvm_attach_attr((void *) &option_attr, &rvm_ipcr, rvm_mpcr)
                != RVM_OK)
        {
                ex_callprint(EX_NUMBER(RVM, RVM_INTERNAL_ERR), EX_INTOK,
301);
 return FALSE;
        }
/* register the IPCR */
if (rvm_stmt_level_register(rvm_mpcr, &rvm_ipcr)
  == RVM_ERROR)
{
 ex_callprint(EX_NUMBER(RVM, RVM_INTERNAL_ERR), EX_INTOK, 302);
 return FALSE;
}
/* Dispatch the check to the provided RVM. */
        if (!rvm_process_dispatch(rvm_mpcr))
        {
  return FALSE;
        }
 return TRUE;
}
/*
** ENCR__CRT_SYMKEY_DIGEST
**
** This routine calls other routines to do the following:
**  - get the system default password from sysattrbutes and
**   decrypt it
**  - generate random salt and create a static key by hashing
**   the password
**  - generate the symmetric column encryption key
**  - encrypt the column encryption key with the hashed password
**
** Parameters:
**  keyname - name of the key to be created
**  keyname_len - length of name of key
**  keylength - length of key in bits
```

```
**  ciphertext - will contain the encrypted symmetric key
**    Expected to be EK_ENCRYPT_VALUE_LEN size
**  dbid - id of database in which key is to be created
**  user_password - TRUE if user password specified
**    FALSE if system encryption password is to be
**    used
**  pwd - user password/system encr password
**  pwdlen - Length of password
**  vers_salt - (out) buffer for generated version/salt
**
** Returns:
**  ex_raise() if there is a problem and returns FALSE.
**
** Side Effects:
**
*/
SYB_STATIC SYB_BOOLEAN
encr__crt_symkey_encrypt(char *keyname, int keyname_len, int keylength,
   BYTE *ciphertext, dbid_t dbid,
   SYB_BOOLEAN user_password, BYTE *pwd, int pwdlen,
   BYTE *vers_salt)
{
 EN_GLOBALCTX  *encrGctx;
 BYTE   keybuf[EK_MAX_SYMKEY_VALUE_LEN]; /* for up
     ** to 256-bit key concat with salt */
 char   *errdesc;
 SYB_BOOLEAN  retval;
 size_t   buflen;
 BYTE   keystatic[EN_AES_DIGEST_LEN]; /* to mix key */
 size_t   kstaticlen; /* Length of static key bytes */
 BYTE   *saltp;     /* Pointer to just the salt */
 short   vers;     /* Version of static key */
#if SANITY
 char   outbuf[EK_ENCRYPT_VALUE_LEN*2+3];
 int   san_i;
#endif /* SANITY */
 encrGctx = Kernel->kencr_ctx;
 retval = TRUE;
 kstaticlen = EN_AES_DIGEST_LEN;
 vers = EK_STATIC_VERS;
 MEMMOVE(&vers, &vers_salt[0], ENCR_VERSION_LEN);
 saltp = &vers_salt[2];
 /* Generate random "salt" */
 if ((en_generateRandomData(encrGctx, saltp, (size_t)ENCR_SALT_LEN))
    != SUCCEED)
 {
  ex_callprint(EX_NUMBER(
   ENCRYPTION,ENCR_BAD_RANDOM_GEN), EX_INTOK, 2);
  return FAIL;
 }
 /*
 ** Make a static key out of the system encryption password/user
 ** password, the generated random salt and static ingredient.
 */
 if (!encr_make_static_key(EK_UKEY, EK_STATIC_VERS, (BYTE *)&pwd[0],
    pwdlen, saltp, ENCR_SALT_LEN, &keystatic[0],
    &kstaticlen))
 {
  /* ENCRCOLS_RESOLVE: Give error ? */
  return FAIL;
 }
#if SANITY
 if (TRACECMDLINE(ENCRYPTION,2))
 {
  sprintf(outbuf, "0x");
  for (san_i = 0; san_i < kstaticlen; san_i++)
  {
   sprintf(outbuf+(san_i*2)+2, "%02x",
```

```
    keystatic[san_i]);
   }
   *(outbuf+(kstaticlen*2 + 2)) = '\0';
   TRACEPRINT("Static Key: %s\n", outbuf);
  }
#endif /* SANITY */
  /*
  ** Call encryption API to create a symmetric key for
  ** keyname
  */
  /*
  ** To create the key, pass in length big enough for the key,
  ** keylength/BITS_PER_BYTE.
  */
  buflen = keylength/BITS_PER_BYTE;
  if (!en_aes_createsymkey(encrGctx, (size_t)keylength, 0,
     keybuf, &buflen, &errdesc))
  {
   /* give error that key creation failed */
   ex_callprint(
    EX_NUMBER(ENCRYPTION, ENCR_CRTKEYFAIL), EX_INTOK, 1, keyname_len, key
name);
    return FALSE;
  }
#if SANITY
  if (TRACECMDLINE(ENCRYPTION,2))
  {
   sprintf(outbuf, "0x");
   for (san_i = 0; san_i < buflen; san_i++)
   {
    sprintf(outbuf+(san_i*2)+2, "%02x",
     keybuf[san_i]);
   }
   *(outbuf+(buflen*2 + 2)) = '\0';
   TRACEPRINT("Sym Key: %s\n", outbuf);
  }
#endif /* SANITY */
  /*
  ** Encrypt the column encryption key and the salt with the
  ** static key.
  */
  if (!encr_encrypt_key_n_salt(&keystatic[0], saltp, keybuf, keylength,
      EK_ENCRYPT_VALUE_LEN, ciphertext))
  {
   retval = FALSE;
  }
  /* Add sentinel byte to salt and key for saving to disk */
  vers_salt[ENCR_VERS_SLT_LEN] = 1;
  keybuf[keylength] = 1;
  MEMZERO(keybuf, EN_AES_KEY_BUFLEN);
  return(retval);
}
/*
** ENCR__GET_SYS_PASSWD
**
*** This routine gets the system dafault password from sysattributes
** and calls the function to decrypt it.
**
** Parameters:
**  keyname  (In) -  Name of key to be created or NULL
**  keyname_len (In) -  Length of keyname or 0
**  keyid  (In) -  Alternative to keyname
**  dbid  (In) -  Id of database where key is to
**     created
**  pwd  (Out) - Buffer for plain password
**  pwdlen  (Out) - Length of outgoing password
**
** Returns:
```

```
**  ex_callprint() if there is a problem and returns FALSE.
**
** Side Effects:
**
*/
SYB_BOOLEAN
encr_get_sys_passwd(char *keyname, int keyname_len, objid_t keyid,
   dbid_t dbid, BYTE *pwd, int *pwdlen)
{
 LOCALPSS(pss);
 ATTRINFO sysattr_args;  /* Arguments passed to
     ** sysattributes.
     */
 char   kname[MAXNAME]; /* For error reporting */
 char  dbname[MAXNAME];/* For error reporting */
 int  dbnlen;  /* For error reporting */
 char  *knamep;
 suid_t  kuid;
 SYB_BOOLEAN changedb; /* TRUE-->need to change dbs */
 *pwdlen = EN_MAXPWDLEN;
 if (changedb = (pss->pcurdb != dbid) ? TRUE : FALSE)
 {
  if ((usedb((BYTE *)NULL, dbid, pss->psuid)) == INVALDBID)
  {
   dbname[0] = '\0';
   (void)getdbname(dbid, (BYTE *)dbname, &dbnlen);
   ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_NODB),EX_USER,1,
     dbnlen, dbname, (int)keyid);
   return FALSE;
  }
 }
 /*
 ** Get system encryption password from sysattributes.
 */
 attrib_initstruct(&sysattr_args);
 sysattr_args.aiclass = COL_ENCRYPT_CLASS;
 sysattr_args.aiattrib = SYSTEM_ENCR_PASSWD;
 strncpy((char *)sysattr_args.aitype, ATTR_TYPE_ENCRCOLS,
   sizeof(sysattr_args.aitype));
 /* Search for matching row */
 if (attrib_getrow(&sysattr_args, pss->pdbtable) != ATTR_ROW_FOUND)
 {
  knamep = keyname;
  /* give error */
  if (keyname_len == 0)
  {
   SYB_ASSERT(keyid != 0);
   get_name(keyid, dbid, (BYTE *)kname, &keyname_len,
    &kuid);
   knamep = kname;
  }
  ex_callprint(
   EX_NUMBER(ENCRYPTION, ENCR_NO_SYSPASSWD),EX_MISSING,1,
    keyname_len, knamep, (int)dbid);
  if (changedb)
  {
   closedb(USEPREV);
  }
  return FALSE;
 }
 /*
 ** Decrypt the system encryption password
 */
 if (ea_decrypt_syspasswd(EK_SPASS_CAT, EK_STATIC_VERS,
  (char *)&sysattr_args.aicharvalue[0],
  sysattr_args.aicharvlen, pwd, pwdlen) == FAIL)
 {
  /* Error already given */
```

```
  if (changedb)
  {
   closedb(USEPREV);
  }
  return FALSE;
 }
 if (changedb)
 {
  closedb(USEPREV);
 }
 return TRUE;
}
/*
** ENCR_MAKE_STATIC_KEY
**
** Used to generate static keys for a given version and a given static
** key type.
**
** Parameters:
**  ktype    (In) - Type of static key requested
**  vers     (In) - Version of key
**  src1     (In) - First source of data for mix
**  src1len    (In) - Length of src1
**  src2     (In) - Second source of data for mix
**  src2len    (In) - Length of src2
**  retbytes   (Out) - Buffer holding digest
**  retblen    (Out) - Length of digest
**
** Returns:
**  TRUE - OK
**  FALSE - Not OK
**
*/
SYB_BOOLEAN
encr_make_static_key(int ktype, int vers, BYTE *src1, int src1len,
  BYTE *src2, int src2len, BYTE *retbytes, size_t *retblen)
{
 EN_GLOBALCTX *encrGctx;
 int  keyindex; /* For internal key data */
 int  keyoffset; /* For internal key data */
 BYTE  *tmp1data; /* Internally gen'd key byte string */
 int  tmp1len;
 BYTE  *tmp2data; /* Internally gen'd key byte string */
 int  tmp2len;
 BYTE  *tmp3data; /* Internally gen'd key byte string */
 int  tmp3len;
 encrGctx = Kernel->encr_ctx;
 SYB_ASSERT(ktype <= EK_UKEY);
 SYB_ASSERT(vers == EK_STATIC_VERS);
 /* Currently all callers supply at least one source of data */
 SYB_ASSERT(src1len > 0);
 if (src1len)
 {
  tmp1data = src1;
  tmp1len = src1len;
 }
 else
 {
  tmp1data = NULL;
  tmp1len = 0;
 }
 if (src2len)
 {
  tmp2data = src2;
  tmp2len = src2len;
 }
 else
 {
```

```
 keyindex = en_ind_tab[vers][ktype].ek2_ind;
 keyoffset = en_ind_tab[vers][ktype].ek2_off;
 tmp2data = get1buf(keyindex, keyoffset);
 tmp2len = EN_INTRNL_KPARTS_LEN;
}
keyindex = en_ind_tab[vers][ktype].ek3_ind;
keyoffset = en_ind_tab[vers][ktype].ek3_off;
tmp3data = get2buf(keyindex, keyoffset);
tmp3len = EN_INTRNL_KPARTS_LEN;
keyindex = 0;
keyoffset = 0;
/*
** Generate static key.
** ENCRCOLS_RESOLVE: Eventually pass in version, which
** will allow us to associate new mixing algorithms with
** a version.
*/
if (!GEN_STATIC_ENCR_KEY(encrGctx, tmp1data, tmp1len, tmp2data,
   tmp2len, tmp3data, tmp3len, retbytes,
   retblen))
{
  return FALSE;
 }
 SYB_ASSERT(*retblen >= EN_AES_DIGEST_LEN);
 return TRUE;
}
#else /* USE_SECURITYBUILDER */
void
encr_crtkey(E_STEP *estep)
{
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 5);
 ex_raise(ENCRYPTION, EX_ANY, EX_CONTROL, 0);
}
SYB_BOOLEAN
encr_make_static_key(int ktype, int vers, BYTE *src1, int src1len,
  BYTE *src2, int src2len, BYTE *retbytes, size_t *retblen)
{
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 8);
 ex_raise(ENCRYPTION, EX_ANY, EX_CONTROL, 0);
}
#endif /* USE_SECURITYBUILDER */
// encols.c
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
/*
** ENCOLS.C
**
** Encrypted columns run-time functions
*/
/* Generic Includes */
# include       <port.h>        /* this MUST be first */
# include       <syb_std.h>     /* include second */
# include       <dtypes.h> /* include third */
# include       <server.h> /* include forth */
/*
** Specific Includes
*/
# include <derror.h>
# include  <src_dcl.h>
# include  <tokens.h>
# include <datatype.h>
/* for trees.h */
# include  <tokenop.h>
# include  <object.h>
# include  <session.h>
# include  <trees.h>
/* for pss.h */
# include  <datetime.h>
# include  <lock.h>
```

```
# include  <lockmgr.h>
# include  <exception.h>
# include  <translate.h>
# include  <loginrec.h>
# include  <foucvt.h>
# include  <dtypes.h>
/* for trace.h */
# include  <pss.h>
# include  <bitbyte.h>
# include  <trace.h>
# include <kernel.h>
# include <encryptkey.h>
# include <encryption.h>
# include <encrypterr.h>
extern STORAGE_FUNCS *Master_xlate;
/*
** Macros
*/
#define PAD_BYTES(_datalen) \
 (EN_AES_BLOCKSIZE - (_datalen % EN_AES_BLOCKSIZE))
/* Private function declarations */
SYB_STATIC void encrcol__xlate PROTO((CONSTANT *));
/* Public function common to all platforms */
/*
** ENCR_KEY_SCHEMA_CHANGE
**
** Check the keys to be used in the current statement to see
** if any have been modified or re-encrypted.  Compare the
** key's schema count2 in sysobjects with the schema count
** saved in the runtime structure.
**
** Parameters:
** encrkp     ptr to list of structures containing key/encryption info
**
** Returns:
** TRUE - there's been a schema change
** FALSE - no schema change
**
*/
SYB_BOOLEAN
encr_key_schema_change(E_ENCRKEYS *encrkp)
{
 E_ENCRKEYS *ekp;  /* Pointer to info about one key */
 OBJECT  objstruct;
 int16  schemact2;
 ekp = encrkp;
 while (ekp)
 {
  if (getobj_crdate_or_schemact(ekp->e_ekid, ekp->e_ekdbid,
    NULL, &schemact2) &&
      ekp->e_ekschemact != schemact2)
  {
   return TRUE;
  }
  ekp = ekp->e_eknext;
 }
 return FALSE;
}
/*
** Public functions useful only to platforms supporting Security Builder
*/
#if USE_SECURITYBUILDER
/*
** COL_ENCRYPT
**
** Using key and encryption information cached in the current
** statement's E_STMT structure, encrypt data and format the
** encrypted column as follows:
```

```
** len  -  if source data has varying length start with a 2 byte
**  length.  Length and data get encrypted together.
** data -  the source data, translated to an independent format,
**  is concatenated to the optional length and padded
**  with random data or zeros so that the length of the
**  plaintext bytes is a multiple of the encryption blocksize
**
** After encryption concatenate optional initialization vector and
** mandatory sentinel byte, to avoid loss of trailing zeros.
**
** Parameters:
** kid     key's id
** kdbid     key's dbid, used with id for identifying cached key
** src     pointer to CONSTANT structure containing plain data
** dest     pointer to CONSTANT structure to hold encrypted data
** buf       pointer to CONSTANT structure that holds padded,
**      source data ready for encryption.
**
** Returns:
** SUCCEED     Encryption succeeded
** FAIL     Something went wrong
**
** Assumptions:
** nothing
**
** Side Effects:
** None.
**
*/
int
col_encrypt(objid_t kid, dbid_t kdbid, CONSTANT *src, CONSTANT *dest,
 CONSTANT *buf)
{
 LOCALPSS(pss);
 STORAGE_FUNCS *sf;  /* Pointer to Master_xlate row */
 E_ENCRKEYS *encrkp; /* Ptr to list of encryption info */
 EN_GLOBALCTX *encrGctx; /* Global encryption context */
 EN_LOCALCTX *encrLctx; /* Local encryption context */
 BYTE  init_vec[EN_AES_BLOCKSIZE];
 BYTE  *ivp;  /* Pointer to iv array */
 int  iv_len;  /* Length of initialization vector */
 BYTE  *plaintp; /* Ptr to plaintext buffer */
 BYTE  *destp;  /* Ptr to destination buffer */
 int  encr_len; /* Length of block to be encrypted */
 int  encr_status; /* Status for encryption setup */
/* ENCR_RESOLVE : Followng decl will be removed when we remove
** 'errbuf' arg from encryption.c API
*/
 char  *dummy;
 int  ret_stat; /* Return status */
#if SANITY
 char  outbuf[EK_ENCRYPT_VALUE_LEN+3];
 int  san_i;
#endif /* SANITY */
 encrGctx = Kernel->kencr_ctx;
 plaintp = buf->value;
 destp = dest->value;
 encr_len = EN_ENCR_VARLEN(src->len, src->type);
 iv_len = 0;
 ivp = (BYTE *)NULL;
 encr_status = EN_ENCRYPT;
 ret_stat = SUCCEED;
 SYB_ASSERT(buf->len >= encr_len);
 /* Look up key and encryption information */
 encrkp = encr_key_lookup(kid, kdbid, pss->pcurstmt);
 SYB_ASSERT(encrkp);
 SYB_ASSERT(dest->maxlen >= (encr_len +
  ((encrkp->e_ekstatus & (EK_INITVECTOR)) ? EN_AES_BLOCKSIZE:0)
```

```
        +1));
/* Key's context (may be null) */
encrLctx = (EN_LOCALCTX *)encrkp->e_ekLctx_enc;
/*
** If the key specifies use of an initialization vector
** generate the random data
*/
if (encrkp->e_ekstatus & (EK_INITVECTOR|EK_RANDOMPAD))
{
 if ((en_generateRandomData(encrGctx, &init_vec[0],
   (size_t)EN_AES_BLOCKSIZE)) != SUCCEED)
 {
  ex_callprint(EX_NUMBER(
      ENCRYPTION,ENCR_BAD_RANDOM_GEN), EX_INTOK, 1);
  return FAIL;
 }
 ivp = &init_vec[0];
 iv_len = EN_AES_BLOCKSIZE;
}
/*
** Use of an init vector implies that each encryption operation
** must use an individual prolog to pass in the vector.
** If no init vector has been specified, we do the prolog
** once per key.  The local context setting tells us whether the
** prolog has been done.
*/
if (encrkp->e_ekstatus & EK_INITVECTOR || (!encrLctx))
{
 if (!encrLctx)
 {
  if (!(encrLctx = ubfalloc(Kernel->kencr_mempool,
    sizeof(EN_LOCALCTX))))
  {
   ex_callprint(EX_NUMBER(
     ENCRYPTION, ENCR_NOMEMORY), EX_RESOURCE, 5);
   ex_raise(ENCRYPTION, ENCR_NOMEMORY,
    EX_CONTROL, 0);
  }
 }
 if (encrkp->e_ekstatus & EK_INITVECTOR)
 {
  encr_status |= EN_INIT_VECTOR;
 }
#if SANITY
 if (TRACECMDLINE(ENCRYPTION,2))
 {
  sprintf(outbuf, "0x");
  for (san_i = 0; san_i < encrkp->e_eklen/BITS_PER_BYTE;
    san_i++)
  {
   sprintf(outbuf+(san_i*2)+2, "%02x",
     *(encrkp->e_ekrawvalue+san_i));
   if (san_i >= EN_AES_KEY_BUFLEN)
   {
    /* Shouldn't happen */
    break;
   }
  }
  *(outbuf+((encrkp->e_eklen/BITS_PER_BYTE)*2 + 2)) = '\0';
  TRACEPRINT("Static Key: %s\n", outbuf);
 }
# endif /* SANITY */
  /* Supply key value and initialization vector */
  if ((en_aes_beginCryptOper(encrGctx, encrLctx,
   encrkp->e_ekrawvalue, encrkp->e_eklen,
   encr_status, ivp, iv_len, &dummy)) != SUCCEED)
  {
   ex_callprint(EX_NUMBER(
```

```
      ENCRYPTION,ENCR_SETUP_FAIL), EX_INTOK, 2);
  return FAIL;
 }
 /* Save context for re-use where no init vector required */
 encrkp->e_ekLctx_enc = (void *)encrLctx;
}
/* Copy varlen (if any) and plain data to static buf */
if (IS_VARLEN_TYPE(src->type))
{
 int16 len;
 len = src->len;
 SWAPSHORT(&len);
 MEMMOVE((BYTE *)&len, plaintp, sizeof(int16));
 plaintp += sizeof(int16);
}
/* Translate data to machine independent format. */
if (ISFLOATTYPE(src->type) || (src->type == INT2) ||
 (src->type == INT4))
{
 encrcol__xlate(src);
}
 /* Move plain data to intermediate buffer */
 MEMMOVE(src->value, plaintp, src->len);
 plaintp += src->len;
 /* Pad with random data or zeros to blocksize */
 if (encrkp->e_ekstatus & EK_RANDOMPAD)
 {
  MEMMOVE(&init_vec[0], plaintp,
   PAD_BYTES((int)(plaintp - buf->value)));
 }
 else
 {
  MEMZERO(plaintp, PAD_BYTES((int)(plaintp - buf->value)));
 }
 /* Encrypt padded data */
 if ((en_aes_encrypt(encrGctx, encrLctx, buf->value,
   (size_t)encr_len, dest->value, &dummy)) != SUCCEED)
 {
  ex_callprint(EX_NUMBER(
   ENCRYPTION,ENCR_ENCRYPTION_FAIL), EX_INTOK, 2);
  ret_stat = FAIL;
 }
#if SANITY
 if (TRACECMDLINE(ENCRYPTION,2))
 {
  sprintf(outbuf, "0x");
  for (san_i = 0; san_i < encr_len; san_i++)
  {
   sprintf(outbuf+(san_i*2)+2, "%02x",
     *(dest->value+san_i));
   if (san_i >= EK_ENCRYPT_VALUE_LEN)
   {
    /* Just give a sampling of decrypted data */
    break;
   }
  }
  *(outbuf+(encr_len*2 + 2)) = '\0';
  TRACEPRINT("Data: %s\n", outbuf);
 }
#endif /* SANITY */
 dest->len = encr_len;
 destp += encr_len;
 if (encrkp->e_ekstatus & EK_INITVECTOR)
 {
  if ((en_aes_endCryptOper(encrGctx, encrLctx, &dummy))
   != SUCCEED)
  {
   ex_callprint(EX_NUMBER(
```

```
        ENCRYPTION,ENCR_END_FAIL), EX_INTOK, 2);
    ret_stat = FAIL;
   }
   if (ret_stat != FAIL)
   {
    /* Append the initialization vector */
    MEMMOVE(&init_vec[0], dest->value + encr_len,
      EN_AES_BLOCKSIZE);
   }
   destp += EN_AES_BLOCKSIZE;
   dest->len += EN_AES_BLOCKSIZE;
  }
  if (ret_stat != FAIL)
  {
   *destp = (BYTE)1;
   dest->len++;
  }
  return ret_stat;
}
/*
** COL_DECRYPT
**
** Using key and encryption information cached in the current
** statement's E_STMT structure, decrypt ciphertext that has the
** following format (looking at it backwards) from the end
** last byte  - a sentinel byte to be stripped off
** one block of bytes (optional): an initialization vector, if the
**  key specifies one
** rest of the ciphertext - the length of this chunk will be
**  a multiple of block size and is the ciphertext to
**  be decrypted.
**
** After decryption, if destination type has varyint length,
** extract length from optional first two bytes to indicate
** length of data.
**
** Parameters:
** kid      key's id
** kdbid      key's dbid, used with id for identifying cached key
** src      pointer to CONSTANT structure containing cipertext
** dest      pointer to CONSTANT structure to hold decrypted data
** buf       pointer to CONSTANT structure that holds copy memory
**
** Returns:
** SUCCEED      Decryption succeeded
** FAIL      Something went wrong
**
** Assumptions:
** nothing
**
** Side Effects:
** None.
**
*/
int
col_decrypt(objid_t kid, dbid_t kdbid, CONSTANT *src, CONSTANT *dest,
 CONSTANT *buf)
{
 LOCALPSS(pss);
 E_ENCRKEYS *encrkp; /* Ptr to list of encryption info */
 EN_GLOBALCTX *encrGctx; /* Global encryption context */
 EN_LOCALCTX *encrLctx; /* Local encryption context */
 BYTE  *init_vp; /* Pointer to init vector */
 int  iv_len;  /* Length of initialization vector */
 BYTE  *plaintp; /* Ptr to intermediate buffer */
 size_t  encr_len; /* Length of decrypted data */
 int  decr_status; /* Status for setting up to decrypt */
/* ENCR_RESOLVE : Followng decl will be removed when we remove
```

```
** 'errbuf' arg from encryption.c API
*/
 char  *dummy;
 int  ret_stat; /* Return status */
#if SANITY
 char  outbuf[EK_ENCRYPT_VALUE_LEN*2+3];
 int  san_i;
#endif /* SANITY */
 encrGctx = Kernel->kencr_ctx;
 plaintp = buf->value;
 init_vp = (BYTE *)NULL;
 iv_len = 0;
 decr_status = EN_DECRYPT;
 ret_stat = SUCCEED;
 /* Look up the key and encryption information */
 encrkp = encr_key_lookup(kid, kdbid, pss->pcurstmt);
 SYB_ASSERT(encrkp);
 /* Key's local context (may be null) */
 encrLctx = (EN_LOCALCTX *)encrkp->e_ekLctx_dec;
 /*
 ** Use of an init vector implies that each decryption operation
 ** must use an individual prolog to pass in the vector.
 ** If no init vector has been specified, we do the prolog
 ** once.  The local context setting tells us whether the
 ** prolog has been done.
 */
 if (encrkp->e_ekstatus & EK_INITVECTOR || (!encrLctx))
 {
  if (!encrLctx)
  {
   if (!(encrLctx = ubfalloc(Kernel->kencr_mempool,
     sizeof(EN_LOCALCTX))))
   {
    ex_callprint(EX_NUMBER(
      ENCRYPTION, ENCR_NOMEMORY), EX_RESOURCE, 6);
    ex_raise(ENCRYPTION, ENCR_NOMEMORY, EX_CONTROL, 0);
   }
  }
  if (encrkp->e_ekstatus & EK_INITVECTOR)
  {
   decr_status |= EN_INIT_VECTOR;
   init_vp = src->value +
    (src->len - (EN_AES_BLOCKSIZE +1));
   iv_len = EN_AES_BLOCKSIZE;
  }
  /* Supply key value and initialization vector */
  if ((en_aes_beginCryptOper(encrGctx, encrLctx,
   encrkp->e_ekrawvalue, encrkp->e_eklen,
   decr_status, init_vp, iv_len, &dummy)) != SUCCEED)
   {
    ex_callprint(EX_NUMBER(
        ENCRYPTION,ENCR_SETUP_FAIL), EX_INTOK, 3);
    return FAIL;
   }
   /* Save context for re-use where no init vector required */
   encrkp->e_ekLctx_dec = (void *)encrLctx;
  }
  encr_len = src->len - 1 - (init_vp ? EN_AES_BLOCKSIZE :0 );
  SYB_ASSERT(buf->len >= encr_len);
#if SANITY
  if (TRACECMDLINE(ENCRYPTION,2))
  {
   sprintf(outbuf, "0x");
   for (san_i = 0; san_i < encr_len; san_i++)
   {
    sprintf(outbuf+(san_i*2)+2, "%02x",
      *(src->value+san_i));
    if (san_i >= EK_ENCRYPT_VALUE_LEN)
```

```
    {
     /* Just give a sampling of decrypted data */
     break;
    }
   }
   *(outbuf+(encr_len*2 + 2)) = '\0';
   TRACEPRINT("%s\n", outbuf);
  }
#endif /* SANITY */
  if (en_aes_decrypt(encrGctx, encrLctx, src->value, encr_len,
   buf->value, &dummy) != SUCCEED)
  {
   ex_callprint(EX_NUMBER(
    ENCRYPTION,ENCR_DECRYPTION_FAIL), EX_INTOK, 1);
   ret_stat = FAIL;
  }
  /* Copy varlen (if any) and plain data to static buf */
  if (IS_VARLEN_TYPE(dest->type))
  {
   int16 len;
   MEMMOVE(buf->value, (BYTE *)&len, sizeof(int16));
   SWAPSHORT(&len);
   dest->len = len;
   plaintp += sizeof(int16);
  }
  else
  {
   dest->len = dest->maxlen;
  }
  MEMMOVE(plaintp, dest->value, dest->len);
  /* Translate INT and FLT data from machine independent format */
  if (ISFLOATTYPE(dest->type) || (dest->type == INT2) ||
   (dest->type == INT4))
  {
   encrcol__xlate(dest);
  }
  if (encrkp->e_ekstatus & EK_INITVECTOR)
  {
   if ((en_aes_endCryptOper(encrGctx, encrLctx, &dummy))
    != SUCCEED)
   {
    ex_callprint(EX_NUMBER(
        ENCRYPTION,ENCR_END_FAIL), EX_INTOK, 3);
    ret_stat = FAIL;
   }
  }
  return ret_stat;
}
/*
** ENCR_KEY_LOOKUP
**
** Using the encryption key's object id and dbid, find the key
** in a list off the current E_STMT and return a pointer to it.
**
** Parameters:
** kdbid     key's dbid, used with id for identifying cached key
** kid     key's id
** estmt     pointer to current estmt
**
** Returns:
** ptr to E_ENCRKEYS struct containing key and encryption info.
**
** Assumptions:
** None
**
** Side Effects:
** None.
**
```

```
*/
E_ENCRKEYS *
encr_key_lookup(objid_t kid, dbid_t kdbid, E_STMT *estmt)
{
 E_ENCRKEYS  *encrkp;
 /* Look for the key and encryption information off the E_STMT */
 if (estmt->e_encrkeys == (E_ENCRKEYS *)NULL)
 {
  return (E_ENCRKEYS *)NULL;
 }
 for (encrkp = estmt->e_encrkeys; encrkp;
     encrkp = encrkp->e_eknext)
 {
  if (kid == encrkp->e_ekid && kdbid == encrkp->e_ekdbid)
  {
   break;
  }
 }
 return(encrkp);
}
/*
** S_DECRYPT_KEYS
**
** Traverse a list of the current statement's encryption keys,
** decrypting each key and opening the encryption context if
** encryption does not use an initialization vector.
**
** Parameters:
** encrkp     ptr to list of structures containing key/encryption info
**
** Returns:
** TRUE: Everything succeeded
** FALSE: Something failed. Should be regarded as fatal by caller
**
** Assumptions:
** None
**
** Side Effects:
** None.
**
*/
SYB_BOOLEAN
s_decrypt_keys(E_ENCRKEYS *encrkp)
{
 EN_GLOBALCTX *encrGctx; /* Global encryption context */
 EN_LOCALCTX *encrLctx; /* Local encryption context */
 int  syspwdlen; /* Length of system password */
 BYTE  syspwd[EN_MAXPWDLEN]; /* Password buffer */
 BYTE  kekbuf[EN_AES_DIGEST_LEN]; /* Key-encrypting key */
 size_t  keklen;  /* Length of key-encrypting key */
 E_ENCRKEYS *ekp;  /* Pointer to info about one key */
 int  ret_stat;
 char  *dummy;
 BYTE  salt[ENCR_SALT_LEN]; /* For validation of KEK */
 short  vers;  /* Version of static key */
#if SANITY
 char  outbuf[EK_ENCRYPT_VALUE_LEN+3];
 int  san_i;
#endif /* SANITY */
 syspwdlen = 0;
 keklen = 0;
 ret_stat = FAIL;
 encrGctx = Kernel->kencr_ctx;
 ekp = encrkp;
 if (!(encrLctx = ubfalloc(Kernel->kencr_mempool,
   sizeof(EN_LOCALCTX))))
 {
  ex_callprint(EX_NUMBER(
```

```
  ENCRYPTION, ENCR_NOMEMORY), EX_RESOURCE, 7);
 ex_raise(ENCRYPTION, ENCR_NOMEMORY, EX_CONTROL, 0);
}
while (ekp)
{
 /* Key encrypted with system password? */
 if (ekp->e_ekstatus & EK_SYSENCRPASS)
 {
  /*
  ** If we haven't done so already, decrypt the
  ** system password and make key encrypting key.
  */
  if (syspwdlen == 0)
  {
   if (!encr_get_sys_passwd((char *)NULL,
    0, ekp->e_ekid, ekp->e_ekdbid,
    &syspwd[0], &syspwdlen))
   {
    /* Error already reported */
    goto fail;
   }
  }
  /* Use password to create the KEK */
  keklen = EN_AES_DIGEST_LEN;
  /* Assert existence of sentinel byte */
  SYB_ASSERT(ekp->e_ekpasswd[ENCR_VERS_SLT_LEN] == 1);
  MEMMOVE(&ekp->e_ekpasswd[ENCR_VERSION_LEN], salt,
   ENCR_SALT_LEN);
  SYB_ASSERT(sizeof(vers) == ENCR_VERSION_LEN);
  vers = GETSHORT(&ekp->e_ekpasswd[0]);
  if (!encr_make_static_key(EK_UKEY, (int)vers,
   (BYTE *)&syspwd[0], syspwdlen, salt,
   ENCR_SALT_LEN, &kekbuf[0], &keklen))
  {
   /* Error already reported */
   goto fail;
  }
  SYB_ASSERT(keklen == EN_AES_DIGEST_LEN);
#if SANITY
  if (TRACECMDLINE(ENCRYPTION,2))
  {
   sprintf(outbuf, "0x");
   for (san_i = 0; san_i < keklen; san_i++)
   {
    sprintf(outbuf+(san_i*2)+2, "%02x",
        kekbuf[san_i]);
   }
   *(outbuf+(keklen*2 + 2)) = '\0';
   TRACEPRINT("Static Key: %s\n", outbuf);
   sprintf(outbuf, "0x");
   for (san_i = 0; san_i < ENCR_SALT_LEN; san_i++)
   {
    sprintf(outbuf+(san_i*2)+2, "%02x",
        salt[san_i]);
   }
   *(outbuf+(ENCR_SALT_LEN*2 + 2)) = '\0';
   TRACEPRINT("Salt: %s\n", outbuf);
  }
#endif /* SANITY */
  /* Validate KEK and decrypt CEK */
  if (!encr_decrypt_key_n_salt(&kekbuf[0], salt,
   ekp->e_ekencrvalue, ekp->e_eklen,
   EN_AES_KEY_BUFLEN, ekp->e_ekrawvalue))
  {
   /* Error already reported */
   goto fail;
  }
 }
```

```
   else /* Key encrypted by user password */
   {
    /* Not yet implemented */
    SYB_ASSERT(0);
   }
   ekp = ekp->e_eknext;
  }
  ekp = encrkp;
  ret_stat = SUCCEED;
fail:
  ubffree(Kernel->kencr_mempool, (void *)encrLctx);
  if (syspwdlen > 0)
  {
   /* Zero out system password */
   MEMZERO(&syspwd[0], syspwdlen);
  }
  if (ret_stat == FAIL)
  {
   /* Zero out raw key values */
   s_clean_encrkeys(encrkp);
  }
  return (ret_stat == SUCCEED) ? TRUE:FALSE;
}
/*
** S_CLEAN_ENCRKEYS
**
** Traverse a list of the current statement's encryption keys,
** zeroing out memory holding private key values and closing
** the encryption context if encryption does not use an initialization
** vector.
**
** Parameters:
** encrkp     ptr to list of structures containing key/encryption info
**
** Returns:
** TRUE: Everything succeeded
** FALSE: Something failed
**
** Assumptions:
** None
**
** Side Effects:
** None.
**
*/
void
s_clean_encrkeys(E_ENCRKEYS *encrkp)
{
 EN_GLOBALCTX *encrGctx;
 char  *dummy;
 encrGctx = Kernel->kencr_ctx;
 while (encrkp)
 {
  MEMZERO(encrkp->e_ekrawvalue, EN_AES_KEY_BUFLEN);
  if (encrkp->e_ekLctx_enc)
  {
   (void)en_aes_endCryptOper(encrGctx,
     (EN_LOCALCTX *)encrkp->e_ekLctx_enc,
     &dummy);
   ubffree(Kernel->kencr_mempool, encrkp->e_ekLctx_enc);
   encrkp->e_ekLctx_enc = (E_ENCRKEYS *)NULL;
  }
  if (encrkp->e_ekLctx_dec)
  {
   (void)en_aes_endCryptOper(encrGctx,
     (EN_LOCALCTX *)encrkp->e_ekLctx_dec,
     &dummy);
   ubffree(Kernel->kencr_mempool, encrkp->e_ekLctx_dec);
```

```
    encrkp->e_ekLctx_dec = (E_ENCRKEYS *)NULL;
   }
   encrkp = encrkp->e_eknext;
  }
 }
}
/*
** ENCR_DECRYPT_KEY_N_SALT
**
** Given a static key-encrypting-key and a symmetrically encrypted
** key decrypt the latter with the former.  Remove the salt after
** decryption and compare with salt parameter to validate the
** integrity of the static key.
**
** A note on how encrypted keys are stored in sysencryptkeys: a
** key stored in sysencryptkeys consists of the key (128, 192,
** or 256 bits) appended with 8 bytes of "salt" rounded up to a
** block boundary and encrypted.  A sentinel byte is appended
** to the cipherdata written to sysencryptkeys.ekvalue.
**
** Parameters:
** static_key (in) ptr to bytes representing kek (bitsize
**       EN_SYSTEM_BIT_KEYSIZE)
** salt  (in) ptr to EN_SALT_BYTE_SIZE bytes of "salt" for
**       verifying decryption of key
** cipherbuf (in) ptr to symmetrically-encrypted key
** keysize  (in) bit size of key, needed because encryption
**       of key may have been padded up.
** plainbuflen (in)  Prevent buffer overruns.
** plainbuf (out) buffer to hold decrypted key or password,
**       with salt & sentinel stripped off.
**
** Returns:
** TRUE: Everything succeeded
** FALSE: Something failed
**
** Assumptions:
** None
**
** Side Effects:
** None.
**
*/
SYB_BOOLEAN
encr_decrypt_key_n_salt(BYTE *static_key, BYTE *salt, BYTE *cipherbuf,
   int keysize, int plainbuflen, BYTE *plainbuf)
{
 EN_GLOBALCTX *encrGctx; /* Global encryption context */
 EN_LOCALCTX *encrLctx; /* Local encryption context */
 char  *dummy;
 SYB_BOOLEAN ret_status;
 BYTE  *saltp;  /* Pointer to salt */
 BYTE  keysaltbuf[EK_MAX_SYMKEY_VALUE_LEN];
     /* Decrypted key and salt */
 size_t  encr_len; /* Derive length from keysize */
#if SANITY
 char  outbuf[EK_ENCRYPT_VALUE_LEN+3];
 int  san_i;
#endif /* SANITY */
 encrGctx = Kernel->kencr_ctx;
 if (!(encrLctx = ubfalloc(Kernel->kencr_mempool,
   sizeof(EN_LOCALCTX))))
 {
  ex_callprint(EX_NUMBER(
   ENCRYPTION, ENCR_NOMEMORY), EX_RESOURCE, 3);
  ex_raise(ENCRYPTION, ENCR_NOMEMORY, EX_CONTROL, 0);
 }
 ret_status = FALSE;
 encr_len = EK_SYMKEY_ENCR_LEN(keysize);
```

```
if (plainbuflen < keysize/BITS_PER_BYTE)
{
 /*
 ** ENCRCOLS_RESOLVE: Buffer overrun - need errormsg?
 */
 goto fail;
}
/* Apply static key (KEK) and context for decryption */
if (!en_aes_beginCryptOper(encrGctx, encrLctx,
  static_key, EN_SYSTEM_BIT_KEYSIZE,
  EN_DECRYPT, NULL, 0, &dummy))
{
 ex_callprint(EX_NUMBER(
  ENCRYPTION,ENCR_SETUP_FAIL),EX_INTOK,4);
 goto fail;
}
/*
** Decrypt concatenation of password and salt into local
** buffer
*/
if (!en_aes_decrypt(encrGctx, encrLctx, cipherbuf, encr_len,
       &keysaltbuf[0], &dummy))
 {
  ex_callprint(EX_NUMBER(
    ENCRYPTION,ENCR_DECRYPTION_FAIL),EX_INTOK,2);
  goto fail;
 }
 /* Clean up context after decryption */
 if (!en_aes_endCryptOper(encrGctx, encrLctx, &dummy))
 {
  ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_END_FAIL),EX_INTOK, 4);
  goto fail;
 }
#if SANITY
 if (TRACECMDLINE(ENCRYPTION,2))
 {
  sprintf(outbuf, "0x");
  for (san_i = 0; san_i < encr_len; san_i++)
  {
   sprintf(outbuf+(san_i*2)+2, "%02x",
    keysaltbuf[san_i]);
  }
  *(outbuf+(encr_len*2 + 2)) = '\0';
  TRACEPRINT("Sym Key: %s\n", outbuf);
 }
#endif /* SANITY */
 /* Isolate salt from decrypted key */
 saltp = &keysaltbuf[keysize/BITS_PER_BYTE];
#if SANITY
if (TRACECMDLINE(ENCRYPTION,2))
{
 sprintf(outbuf, "0x");
 for (san_i = 0; san_i < ENCR_SALT_LEN; san_i++)
 {
  sprintf(outbuf+(san_i*2)+2, "%02x",
   saltp[san_i]);
 }
 *(outbuf+(ENCR_SALT_LEN*2 + 2)) = '\0';
 TRACEPRINT("saltp: %s\n", outbuf);
 sprintf(outbuf, "0x");
 for (san_i = 0; san_i < ENCR_SALT_LEN; san_i++)
 {
  sprintf(outbuf+(san_i*2)+2, "%02x",
   salt[san_i]);
 }
 *(outbuf+(ENCR_SALT_LEN*2 + 2)) = '\0';
 TRACEPRINT("salt: %s\n", outbuf);
}
```

```
#endif /* SANITY */
 if (!MEM_EQ(saltp, ENCR_SALT_LEN, salt, ENCR_SALT_LEN))
 {
  /*
  ** If the salt encrypted with the key doesn't
  ** match the external salt, probably means that
  ** the password used to generate the static key
  ** is wrong.
  */
  ex_callprint(EX_NUMBER(
   ENCRYPTION,ENCR_KEY_DECRYPTION),EX_USER,1);
  goto fail;
 }
 /* Move just key value into return buffer */
 MEMMOVE(keysaltbuf, plainbuf, keysize/BITS_PER_BYTE);
 ret_status = TRUE;
fail:
 ubffree(Kernel->kencr_mempool, (void *)encrLctx);
 return ret_status;
}
/*
** ENCR_ENCRYPT_KEY_N_SALT
**
** Given a static key-encrypting-key, a raw key and salt,
** concatenate the raw bytes and salt and encrypt with the KEK.
** Add a sentinel byte after encryption.
**
** Parameters:
** static_key (in) ptr to EN_SYSTEM_BIT_KEYSIZE bits representing
**       KEK
** salt  (in) ptr to EN_SALT_BYTE_SIZE bytes of "salt" for
**       concatenation to plaintext before encryption
** rawkey  (in) ptr to raw key to be encrypted
** keysize  (in) Keysize in bits
** ciphblen (in) Prevent buffer overruns.
** cipherbuf (out) ptr to results of encryption.
**
** Returns:
** TRUE: Everything succeeded
** FALSE: Something failed
**
** Assumptions:
** None
**
** Side Effects:
** None.
**
*/
SYB_BOOLEAN
encr_encrypt_key_n_salt(BYTE *static_key, BYTE *salt, BYTE *rawkey,
   int keysize, int ciphbuflen, BYTE *cipherbuf)
{
 EN_GLOBALCTX *encrGctx; /* Global encryption context */
 EN_LOCALCTX *encrLctx; /* Local encryption context */
 char  *dummy;
 SYB_BOOLEAN ret_status;
 BYTE  key_salt_buf[EK_MAX_SYMKEY_VALUE_LEN];
     /* Concatenation of raw key & salt */
 int  key_bytes; /* Keysize in bytes */
 encrGctx = Kernel->kencr_ctx;
 if (!(encrLctx = ubfalloc(Kernel->kencr_mempool,
   sizeof(EN_LOCALCTX))))
 {
  ex_callprint(EX_NUMBER(
   ENCRYPTION, ENCR_NOMEMORY), EX_RESOURCE, 4);
  ex_raise(ENCRYPTION, ENCR_NOMEMORY, EX_CONTROL, 0);
 }
 ret_status = FALSE;
```

```
if (ciphbuflen < (EK_SYMKEY_ENCR_LEN(keysize) + 1))
{
 /*
 ** ENCRCOLS_RESOLVE: Do we need errors for
 ** internal buffer overrun problems?
 ** ex_callprint("An internal buffer required during
 ** an encryption operation is too small.  This is an
 ** internal error."
 */
 goto fail;
}
/* Apply static key (KEK) and context for encryption */
if (!en_aes_beginCryptOper(encrGctx, encrLctx,
  static_key, EN_SYSTEM_BIT_KEYSIZE,
  EN_ENCRYPT, NULL, 0, &dummy))
{
 ex_callprint(EX_NUMBER(
  ENCRYPTION,ENCR_SETUP_FAIL),EX_INTOK,5);
 goto fail;
}
/* Concatenate raw key and salt */
 key_bytes = keysize/BITS_PER_BYTE;
 MEMMOVE(rawkey, key_salt_buf, key_bytes);
 MEMMOVE(salt, key_salt_buf + key_bytes, ENCR_SALT_LEN);
 /* Encrypt key and salt into buffer from caller */
 if (!en_aes_encrypt(encrGctx, encrLctx, key_salt_buf,
   (size_t)EK_SYMKEY_ENCR_LEN(keysize), cipherbuf,
   &dummy))
 {
  ex_callprint(EX_NUMBER(
    ENCRYPTION,ENCR_DECRYPTION_FAIL),EX_INTOK,3);
  goto fail;
 }
 /* Clean up context after encryption */
 if (!en_aes_endCryptOper(encrGctx, encrLctx, &dummy))
 {
  ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_END_FAIL),EX_INTOK, 5);
  goto fail;
 }
 /* Append sentinel byte */
 cipherbuf[EK_SYMKEY_ENCR_LEN(keysize)] = 1;
 ret_status = TRUE;
fail:
 ubffree(Kernel->kencr_mempool, (void *)encrLctx);
 return ret_status;
}
#else /* USE_SECURITYBUILDER */
/* stubs */
int
col_encrypt(objid_t kid, dbid_t kdbid, CONSTANT *src, CONSTANT *dest,
 CONSTANT *buf)
{
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 7);
 return FAIL;
}
int
col_decrypt(objid_t kid, dbid_t kdbid, CONSTANT *src, CONSTANT *dest,
 CONSTANT *buf)
{
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 8);
 return FAIL;
}
E_ENCRKEYS *
encr_key_lookup(objid_t kid, dbid_t kdbid, E_STMT *estmt)
{
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 9);
 return NULL;
}
```

```
SYB_BOOLEAN
s_decrypt_keys(E_ENCRKEYS *encrkp)
{
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 10);
 return FALSE;
}
void
s_clean_encrkeys(E_ENCRKEYS *encrkp)
{
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 11);
}
SYB_BOOLEAN
encr_decrypt_key_n_salt(BYTE *static_key, BYTE *salt, BYTE *cipherbuf,
   int keysize, int plainbuflen, BYTE *plainbuf)
{
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 12);
}
SYB_BOOLEAN
encr_encrypt_key_n_salt(BYTE *static_key, BYTE *salt, BYTE *rawkey,
   int keysize, int ciphbuflen, BYTE *cipherbuf)
{
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 13);
}
#endif /* USE_SECURITYBUILDER */
/*
** Private functions
*/
/*
** ENCRCOL__XLATE
**
** Translate data into canonical format before encryption
** and after decryption. Translation done in place.
**
** Parameters:
** data     ptr to CONSTANT structure containing data
**
** Returns:
** nothing
**
** Side Effects:
** None.
**
*/
SYB_STATIC void
encrcol__xlate(CONSTANT *data)
{
 STORAGE_FUNCS *s;  /* Pointer to Master_xlate row */
 switch (data->type) {
   case INT2:
   SWAPSHORT((int16 *)data->value);
   break;
    case INT4:
   SWAPLONG((int32 *)data->value);
   break;
    case FLT4:
   s = &Master_xlate[FLT4_IEEE_HI];
   (s->tfxlate)((void *)&data->value, sizeof(float));
   break;
    case FLT8:
   s = &Master_xlate[FLT_IEEE_HI];
   (s->tfxlate)((void *)&data->value, sizeof(double));
   break;
 }
}
// encolsadmin.c
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
/*
** Encrypted Columns encr_admin built-in support module
```

```
*/
#include <port.h> /* always required as first sybase include file */
#include <syb_std.h> /* always required as second sybase include file */
#include <dtypes.h> /* always required as third sybase include file */
#include <server.h> /* always required as fourth sybase include file */
/*
** This file is only useful on platforms where Security Builder libs
** are available.  A stub for encr_admin() is provided for the non-porte
d
** platforms.
*/
#include <config.h>
#include <cfg_ds.h>
#include <cfg_def.h>
#include <derror.h>
#include <exception.h>
#include <object.h>
#include <session.h>
#include <tokens.h>
#include <tokenop.h>
#include <parserr.h>
#include <phrases.h>
#include <pss.h>
#include <textmgr.h>
#include <sysattr.h>
#include <catalog.h>
#include <dbtable.h>
#include <xactmgr_internal.h>
#include <bitbyte.h> /* for trace.h */
#include <trace.h> /* for TRACEPRINT() */
#include <rvm_internal.h>
#include <rvm_dcl.h>
#include <rvmerr.h>
#include <src_dcl.h>
#include <memfrg.h>
#include <lock.h>
#include <encryptkey.h>
#include <encryption.h>
#include <password.h>
#include <execerr.h>
#include <encrypterr.h>
/* Length of random salt converted to hex */
#define EASALTHEXLEN ENCR_SALT_LEN*2
/*
** The maximum size buffer passed to the encryption API.  It allows
** for:
** 1 byte length of plain passwd
** up to 64 bytes of password
** appended salt (in hex)
** all rounded up to a multiple of the AES block size
*/
#define EAMAXENCRBUF (((EN_MAXPWDLEN+EASALTHEXLEN)/EN_AES_BLOCKSIZE) +1)
 \
    * EN_AES_BLOCKSIZE
/* The maximum length a hex representation of the encrypted passwd+salt,
** concat'd with hex salt.  When prefixed with '0x' and appended with a
** sentinel byte, this is the way passwords are saved to
** sysattributes.charvalue and how they are expected as input for
** replication.
*/
#define EAMAXHEXPLEN ((EAMAXENCRBUF*2) + EASALTHEXLEN + 3)
/* Types of sp_encryption commands */
#define EN_SET_PASSWD 1
/* Number of locks obtained when modifying system encryption password */
#define NUMLOCKS 2
/* forward references */
SYB_STATIC int ea__setpasswd  PROTO((char *, CS_INT *,
            char *, CS_INT *,
```

```
            CS_INT, CS_INT ,
            int *, int *));
SYB_STATIC int ea__reencrypt_symkeys PROTO((char *, int, char *, int,
      XDES *));
SYB_STATIC SYB_BOOLEAN ea__validate_passwds PROTO((char *, int, char *,
int));
/*
** A permanent table for lookup of indices and offsets into ASE static
** random data for the dynamic construction of internal keys. Uses of
** such keys are for encryption of column encryption keys, encryption
** of the system password for saving to sysattributes and encryption
** of the system password for replication.
*/
extern ENCR_LOOKUP en_ind_tab[2][3] =
{
    /* EK_STATIC_VERS_0 */
    { /* EK_SPASS1_IND 0 */
 { 3,  57,   0,  103,  2, 141},
 /* EK_SPASS2_IND 1 */
 {16, 131,  24,   33,  9,  99},
 /* EK_UKEY_IND 2  */
 { 5,  45,  -1,   -1, -1,  -1}
 /* Add entries for additional static keys here */
    },
    /* Add entries for version 1 here */
    {
 {-1,  -1,  -1,  -1,  -1,  -1},
 {-1,  -1,  -1,  -1,  -1,  -1},
 {-1,  -1,  -1,  -1,  -1,  -1}
    }
};
/*
** EA__SETPASSWD
**
** Description:
** Add encrypted password row to sysattributes or replace existing
** password.  The relevant sysattributes columns are:
**
** class  smallint COL_ENCRYPT_CLASS
** attribute smallint SYSTEM_ENCR_PASSWD
** object_type  char(2)  ATTR_TYPE_ENCRCOLS
** char_value char(255) Encrypted system passwd (hex),
**     concatenated with hex salt
** object_info1 int  version of static key encryption
**
** Parameters:
** newpasswd  the password to set
** lenpnew  pointer to the length of the new password
** oldpasswd the previous (existing) password
** lenpold  pointer to the length of the previous password
** maxnewlen Max length of the buffer for the new password
** maxoldlen Max length of the buffer for the old password
** fmt  ptr to fmt (hex or char) of password strings
** vers  ptr to static encryption version
**
** Side effects:
** Passwords stored in the constants will be stored encrypted for
** replication.
**
** Returns:
** SUCCEED Row added/updated
** FAIL Unable to add/update row
**
*/
SYB_STATIC int
ea__setpasswd(char *newpasswd, CS_INT *lenpnew,
       char *oldpasswd, CS_INT *lenpold,
       CS_INT maxnewlen, CS_INT maxoldlen, int *fmt, int *vers)
```

```
{
#if USE_SECURITYBUILDER
 LOCALPSS(pss);
 TRANPARAMS(xprm);
 ATTRINFO sysattr_args; /* Arguments passed to sysattributes */
 ATTRINFO sysattr_oldargs;/* Used to update sysattributes */
 int  buflen;  /* Length of encrypted password */
 BYTE  buf[EAMAXHEXPLEN+1];/* To hold encrypted password
     ** plus null term */
 BYTE  *bufp;  /* Ptr to encrypted passwd */
 int  en_vers;
 int  retstat;
 int  newplen; /* New passwd length */
 int  oldplen; /* Old passwd length */
 VOLATILE struct
 {
  PSS *pss;
  XDES *xdes;
  BYTE pwd[EN_MAXPWDLEN]; /* to hold clear password */
  int pwdlen;  /* Pass length of pwd */
  BYTE npwd_x[EN_MAXPWDLEN]; /* xlate hex new password */
  int npwdlen;
  BYTE opwd_x[EN_MAXPWDLEN]; /* xlate hex old password */
  int opwdlen;
 } copy;
SYB_NOOPT(copy);
MEMZERO(&copy, sizeof(copy));
copy.pss = pss;
copy.xdes = (XDES*) NULL;
bufp = &buf[0];
retstat = FAIL;
en_vers = (vers != NULL) ? *vers : EK_STATIC_VERS;
/* store the current length of passwords in local variables */
newplen = *lenpnew;
oldplen = *lenpold;
/* Exception handling and backout section */
if (ex_handle(EX_ANY, EX_ANY, EX_ANY, (EXC_FUNC_PTR) hdl_backout_msg))
{
 if (copy.xdes)
 {
  copy.xdes = (XDES*) NULL;
  MEMZERO(&copy.pwd, copy.pwdlen);
  MEMZERO(&copy.opwd_x, copy.opwdlen);
  MEMZERO(&copy.npwd_x, copy.npwdlen);
  XACTPRM_END(xprm, NULL, 0, copy.pss, XACT_LOCAL);
  xact_rollback(&xprm);
 }
 goto fail;
}
/*
** Decrypt old and new passwords passed in hex format.
*/
if ((fmt != NULL) && (*fmt == EK_HEX_SYSENCRPASSWD))
{
 copy.npwdlen = EN_MAXPWDLEN;
 if ((newplen) && (ea_decrypt_syspasswd(EK_SPASS_REP, en_vers,
   newpasswd, newplen, (BYTE *)&copy.npwd_x[0],
   (int *)&copy.npwdlen) == FAIL))
 {
  /* Error already given */
  goto fail;
 }
 /* Copy deciphered password to the buffer storing this param */
 if (newplen)
 {
  newplen = copy.npwdlen;
  MEMMOVE((BYTE *) &copy.npwd_x[0],
   (BYTE *) newpasswd, MIN(newplen, maxnewlen));
```

```
 }
 copy.opwdlen = EN_MAXPWDLEN;
 if ((oldplen) && (ea_decrypt_syspasswd(EK_SPASS_REP, en_vers,
   oldpasswd, oldplen, (BYTE *)&copy.opwd_x[0],
   (int *)&copy.opwdlen) == FAIL))
 {
  /* Error already given */
  goto fail;
 }
 /* Copy deciphered password to the buffer storing this param */
 if (oldplen)
 {
  oldplen = copy.opwdlen;
  MEMMOVE((BYTE *) &copy.opwd_x[0],
   (BYTE *) oldpasswd, MIN(oldplen, maxoldlen));
 }
}
/* else ignore other values */
if ((vers != NULL) && ((*vers < 0) || (*vers > EK_STATIC_VERS)))
{
 /* Ignore illegal version */
 goto fail;
}
/* Get minor semantic checks done */
if (!ea__validate_passwds(newpasswd, newplen, oldpasswd,
   oldplen))
{
 goto fail;
}
/*
** See if system encryption password row already exists.
** Set up to search SYSATTRIBUTES in the current database
** for SYSTEM_ENCR_PASSWD.
        */
        attrib_initstruct(&sysattr_args);
        sysattr_args.aiclass = COL_ENCRYPT_CLASS;
sysattr_args.aiattrib = SYSTEM_ENCR_PASSWD;
strncpy((char *)sysattr_args.aitype, ATTR_TYPE_ENCRCOLS,
  sizeof(sysattr_args.aitype));
/* Search for matching row */
        switch (attrib_getrow(&sysattr_args, pss->pdbtable))
{
  case ATTR_ERROR:
     ex_callprint(EX_NUMBER(
  ENCRYPTION,ENCR_SYSPASS_INTERR), EX_INTOK, 1);
 goto fail;
  case ATTR_NOT_FOUND:
 /*
 ** If there is not system encrption password, but an old passwd
 ** has been passed, raise an error.
 */
 if (oldplen != 0)
 {
  ex_callprint(
      EX_NUMBER(ENCRYPTION,ENCR_NO_OLDSYSPASSWD), EX_USER, 1);
  goto fail;
 }
 break;
  case ATTR_ROW_FOUND:
 /* User needs to supply the old password if resetting */
 if (oldplen == 0)
 {
  ex_callprint(EX_NUMBER(
   ENCRYPTION,ENCR_SYSPASSWD_NOT_RESET), EX_USER, 1);
  goto fail;
 }
 /*
 ** Compare supplied old password and length with value
```

```
** in sysattributes.
*/
copy.pwdlen = EN_MAXPWDLEN;
/* First, decrypt password from sysattributes */
if (ea_decrypt_syspasswd(EK_SPASS_CAT, en_vers,
  (char *)&sysattr_args.aicharvalue[0],
  sysattr_args.aicharvlen,
  (BYTE *)&copy.pwd[0],
  (int *)&copy.pwdlen) == FAIL)
{
 /* Error already given */
 goto fail;
}
/* Check match */
if ((copy.pwdlen != oldplen) ||
    STRNCMP(oldpasswd, &copy.pwd[0], oldplen))
{
 ex_callprint(EX_NUMBER(
   ENCRYPTION,ENCR_SYSPASSWD_NOT_RESET), EX_USER, 1);
 goto fail;
}
 /* Error if resetting to same password */
 if (((newplen) != 0 && (newplen == copy.pwdlen)) &&
     !(STRNCMP(newpasswd, &copy.pwd[0], copy.pwdlen)))
 {
  ex_callprint(EX_NUMBER(
   RVM,RVM_SAME_PASSWORD), EX_USER, 2);
  goto fail;
 }
}
/*
** We are here because either the password is being set for the
** first time, or the old password matches.  Encrypt new password
** (if not null) and convert to hex.
*/
buflen = (newplen > 0) ? EAMAXHEXPLEN+1 : 0;
if ((newplen > 0) && !(ea_encrypt_syspasswd(EK_SPASS_CAT, en_vers,
   newpasswd, newplen, (char *)bufp, &buflen)))
{
 ex_callprint(
     EX_NUMBER(ENCRYPTION,ENCR_SYSPASS_CRYPTERR), EX_INTOK, 1,
     TOKENNAME(ENCRYPT));
 goto fail;
}
/* Set up for update transaction. */
XACTPRM_LOCAL(xprm, "$setsysencrpasswd", 16, NULL, pss->pdbtable,
  BEGINXACT_UPDATE);
if (xact_begin(&xprm) != XACTRV_SUCCESS)
{
 ex_callprint(EX_NUMBER(ENCRYPTION, ENCR_SYSPASS_INTERR), EX_INTOK, 2);
 goto fail;
}
copy.xdes = xprm.xdes;
/*
** If re-setting system encryption password, we must re-encrypt
** dependent keys.  The following function will do so and will
** disallow unsetting the password if there are dependent keys.
*/
if ((oldplen > 0) && (ea__reencrypt_symkeys(newpasswd, newplen,
   oldpasswd, oldplen, xprm.xdes)) != SUCCEED)
{
 goto fail;
}
/*
** Set up update or insertion or deletion to sysattributes.
*/
if (buflen == 0)
{
```

```
 /* Deletion of system password */
 attrib_initstruct(&sysattr_args);
 sysattr_args.aiclass = COL_ENCRYPT_CLASS;
 sysattr_args.aiattrib = SYSTEM_ENCR_PASSWD;
 strncpy((char *)sysattr_args.aitype, ATTR_TYPE_ENCRCOLS,
  sizeof(sysattr_args.aitype));
 if (!attrib_delrows(&sysattr_args, copy.xdes))
 {
  ex_callprint(
  EX_NUMBER(ENCRYPTION,ENCR_SYSPASS_INTERR),EX_INTOK,4);
  ex_raise(ENCRYPTION,ENCR_SYSPASS_INTERR,EX_CONTROL,0);
 }
}
else if (oldplen > 0)
{
 /* Replace existing system encryption password. */
        attrib_initstruct(&sysattr_args);
 MEMMOVE(bufp, (BYTE *)sysattr_args.aicharvalue, buflen);
 sysattr_args.aiobjinfo1 = en_vers;
 sysattr_args.aicharvlen = buflen;
        attrib_initstruct(&sysattr_oldargs);
        sysattr_oldargs.aiclass = COL_ENCRYPT_CLASS;
 sysattr_oldargs.aiattrib = SYSTEM_ENCR_PASSWD;
 strncpy((char *) sysattr_oldargs.aitype, ATTR_TYPE_ENCRCOLS,
  sizeof(sysattr_oldargs.aitype));
 if (!attrib_updaterow(sysattr_oldargs, sysattr_args,
   copy.xdes, XMOD_DEFERRED))
 {
  ex_callprint(
  EX_NUMBER(ENCRYPTION,ENCR_SYSPASS_INTERR), EX_INTOK, 3);
  ex_raise(ENCRYPTION,ENCR_SYSPASS_INTERR, EX_CONTROL, 0);
 }
}
else
{
 /*
 ** Insert system password for the first time. Set all
 ** relevant fields.
 */
 MEMMOVE(bufp, (BYTE *)sysattr_args.aicharvalue, buflen);
 attrib_initstruct(&sysattr_args);
 sysattr_args.aicharvlen = buflen;
        sysattr_args.aiclass = COL_ENCRYPT_CLASS;
 sysattr_args.aiattrib = SYSTEM_ENCR_PASSWD;
 sysattr_args.aiobjinfo1 = en_vers;
 strncpy((char *) sysattr_args.aitype, ATTR_TYPE_ENCRCOLS,
  sizeof(sysattr_args.aitype));
 if (!attrib_insrow(&sysattr_args, copy.xdes))
 {
  ex_callprint(
  EX_NUMBER(ENCRYPTION,ENCR_SYSPASS_INTERR), EX_INTOK, 4);
  ex_raise(ENCRYPTION,ENCR_SYSPASS_INTERR, EX_CONTROL, 0);
 }
}
/* If we are setting a new password, encrypt it for replication.  */
if (newplen)
{
 buflen = EAMAXHEXPLEN+1;
 if (!ea_encrypt_syspasswd(EK_SPASS_REP, EK_STATIC_VERS,
   newpasswd, newplen, (char *)bufp, &buflen))
 {
  ex_callprint(
   EX_NUMBER(ENCRYPTION,ENCR_SYSPASS_CRYPTERR),
   EX_INTOK, 1, TOKENNAME(ENCRYPT));
  goto fail;
 }
 /* Be sure the buffer is big enough to hold the value */
 SYB_ASSERT(buflen <= maxnewlen);
```

```
 MEMMOVE((BYTE *)&bufp[0], (BYTE *) &newpasswd[0],
     MIN(buflen, maxnewlen));
 /* Copy new length to the constant of this parameter */
 *lenpnew = buflen;
       }
/* If the oldpasswd was suppliend, encrypt it for replication */
if (oldplen)
{
 buflen = EAMAXHEXPLEN+1;
 if (!ea_encrypt_syspasswd(EK_SPASS_REP, EK_STATIC_VERS,
   oldpasswd, oldplen, (char *)bufp, &buflen))
 {
  ex_callprint(
   EX_NUMBER(ENCRYPTION,ENCR_SYSPASS_CRYPTERR),
   EX_INTOK, 1, TOKENNAME(ENCRYPT));
  goto fail;
 }
 /* Be sure the buffer is big enough to hold the value */
 SYB_ASSERT(buflen <= maxoldlen);
 MEMMOVE((BYTE *) &bufp[0], (BYTE *) &oldpasswd[0],
     MIN(buflen, maxoldlen));
   /* Copy new length to the constant of this parameter */
   *lenpold = buflen;
  }
  retstat = SUCCEED;
  /* Fall through for cleanup */
fail:
  if (copy.xdes)
  {
   (void) xact_commit(&xprm);
   copy.xdes = NULL;
  }
  return retstat;
#else
  ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 3);
  return FALSE;
#endif /* USE_SECURITYBUILDER */
}
/*
** EA__VALIDATE_PASSWDS
**
** Perform semantic checks on system encryption passwords, checking
** length against configured min and max limits and configured
** digit requirement.
**
** Parameters:
** newpasswd  the password to set
** newplen  length of the password
** oldpasswd the previous (existing) password
** oldplen  length of the previous password
**
** Returns:
** TRUE All checks succeeded
** FALSE A check failed
**
*/
SYB_STATIC SYB_BOOLEAN
ea__validate_passwds(char *newpasswd, int newplen, char *oldpasswd,
    int oldplen)
{
#if USE_SECURITYBUILDER
  int  minpwdlen; /* minimum length of password */
  /* Require the password length be within maximum limit */
  if (newplen > EN_MAXPWDLEN)
  {
   ex_callprint(EX_NUMBER(PARSER,P_TOKENTOOLONG), EX_SYNTAX, 131,
    PH_BIGPARAM, 6, "******", EN_MAXPWDLEN);
   return FALSE;
```

```
}
/*
** If ASE is configured to require a minimum password length,
** enforce it, unless the sysstem password is being unset.
*/
minpwdlen = (int32) Resource->rconfig->cfgminpwdlen;
if ((newplen > 0) && (newplen < minpwdlen))
{
 ex_callprint(EX_NUMBER(
  RVM, RVM_SHORT_PASSWORD), EX_PERMIT, 3, minpwdlen);
 return FALSE;
}
/*
** If ASE is configured to require a numerical digit in a
** password, enforce that.
*/
if ((newplen > 0) && (Resource->rconfig->cfgcheckpwddigit == 1))
{
 if (!(check_pwdfor_digit((BYTE *)newpasswd, newplen)))
 {
  ex_callprint(EX_NUMBER(
   EXEC2, SPD_PWD_NODIGIT), EX_USER, 3);
  return FALSE;
 }
}
/*
** If old password has been supplied, check that its length
** is within maximum length.
*/
if (oldplen > EN_MAXPWDLEN)
{
 ex_callprint(EX_NUMBER(PARSER,P_TOKENTOOLONG), EX_USER, 132,
  PH_BIGPARAM, EN_MAXPWDLEN, oldpasswd, EN_MAXPWDLEN);
 return FALSE;
}
return TRUE;
#else
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 8);
 return FALSE;
#endif /* USE_SECURITYBUILDER */
}
/*
** EA__REENCRYPT_SYMKEYS
**
** Description:
** For an updated system encryption password, re-encrypt dependent
** keys.  First, decrypt key using static key derived from old
** password, then re-encrypt with static key derived from new
** password. Disallow removal of system encryption password if
** there are dependent keys.
**
** Parameters:
** newpasswd  the password being set
** newplen  length of the password
** oldpasswd the previous (existing) password
** oldplen  length of the previous password
** xdes  existing transaction descriptor
**
** Returns:
** SUCCEED Keys updated successfully
** FAIL Something went wrong
**
*/
SYB_STATIC int
ea__reencrypt_symkeys(char *newpasswd, int newplen, char *oldpasswd,
   int oldplen, XDES *xdes)
{
#if USE_SECURITYBUILDER
```

```
LOCALPSS(pss);
SDES  *read_sysencrkeys; /* Sdes used for reading */
SDES  *write_sysencrkeys; /* Sdes used for updating */
SDES  *mod_sysobjects; /* Sdes used for sysobjects */
OBJECT  *objrow;  /* Row found in sysobjects */
int16  objschema2;  /* Schema change counter */
LOCKREQUEST lock_requests[NUMLOCKS];/* For lock_multiple() */
LOCKREQUEST *lock_requestsp[NUMLOCKS]; /* For lock_multiple() */
SARG  keys1[1];  /* For search of sysobjects */
int  actual_locks;  /* Number of lock requests */
BYTE  databuf[ENCRK_ROW_BUF_SIZE]; /* Sysencryptkeys row */
ENCRYPTKEY encryptkey;  /* Sysencryptkeys row */
int  rowlen;   /* Len of sysencryptkeys row */
int  lencol[ENCR_VARCOL_COUNT]; /* Array of varlengths */
BYTE  rowbuf[ENCRK_ROW_BUF_SIZE];
BUF  *buf;   /* row from getnext() */
size_t  okeklen;  /* Old key-encrypting key
     ** length */
BYTE  okekbuf[EN_AES_DIGEST_LEN];/* Old key-encrypting
     ** key */
size_t  nkeklen;  /* New kek length */
BYTE  nkekbuf[EN_AES_DIGEST_LEN];/* New kek */
BYTE  plainbuf[EN_AES_KEY_BUFLEN]; /* Decrypted key buf */
BYTE  encrkbuf[EK_MAX_SYMKEY_VALUE_LEN]; /* Re-encrypted
     ** encryption key */
BYTE  *saltp;   /* salt from ekpasswd */
short  vers;   /* version from ekpasswd */
VOLATILE struct
{
 SDES *er_sdes; /* Read sdes for sysencryptkeys */
 SDES *ew_sdes; /* Write sdes for sysencryptkeys */
 SDES *o_sdes; /* sdes for update of sysobjects */
 XDES *xdes;
 int retstat;
} copy;
okeklen = 0;
nkeklen = 0;
MEMZERO(&encryptkey, sizeof(ENCRYPTKEY));
MEMZERO(&copy, sizeof(copy));
copy.retstat = FAIL;
/* Open sysencryptkeys for scanning */
copy.er_sdes = read_sysencrkeys =
 OPEN_SYSTAB_WITH_DBTABLE(SYSENCRYPTKEYS, xdes->xdbptr);
/* Open sysencryptkeys for updating */
copy.ew_sdes = write_sysencrkeys =
 OPEN_SYSTAB_WITH_DBTABLE(SYSENCRYPTKEYS, xdes->xdbptr);
/* Open sysobjects for modifying schema count */
copy.o_sdes = mod_sysobjects =
 OPEN_SYSTAB_WITH_DBTABLE(SYSOBJECTS, xdes->xdbptr);
if (ex_handle(EX_ANY, EX_ANY, EX_ANY, (EXC_FUNC_PTR) hdl_backout_msg))
{
 goto cleanup;
}
read_sysencrkeys->sstat |= (SS_FGLOCK | SS_L1LOCK);
write_sysencrkeys->sstat |= (SS_FGLOCK | SS_UPDLOCK | SS_STMTLOCK);
write_sysencrkeys->read_sdes = read_sysencrkeys;
mod_sysobjects->sstat |= (SS_FGLOCK | SS_UPDLOCK | SS_L1LOCK);
/*
** Get all the locks up front needed for re-encrypting keys.
*/
actual_locks = 0;
LOCKREQ_ARY_SETUP(lock_requestsp, lock_requests, NUMLOCKS);
LOCKREQ_INIT(
 lock_requests[actual_locks], EX_TAB, SYSENCRYPTKEYS,
 pss->pcurdb, LOCKSUFFCLASS_XACT, PCUR_XACTLOCKS(pss),
 LCTX_XACT, actual_locks, NUMLOCKS, 129);
actual_locks++;
LOCKREQ_INIT(
```

```
 lock_requests[actual_locks], EX_TAB, SYSOBJECTS,
 pss->pcurdb, LOCKSUFFCLASS_XACT, PCUR_XACTLOCKS(pss),
 LCTX_XACT, actual_locks, NUMLOCKS, 130);
if (lock_multiple(lock_requestsp, actual_locks) < 0)
{
 ex_raise(EXEC2, SPD_LOCKFAIL, EX_INTOK, 2);
}
/*
** Scan all rows of sysencryptkeys to to detect keys
** encrypted with the system encryption password.
*/
scan_copy_init(read_sysencrkeys, SCAN_COPY_DATA_ROW, databuf,
  (BYTE *)NULL, subst_rcopy, (BYTE *)NULL);
startscan(read_sysencrkeys, SCAN_NOINDEX, SCAN_NORMAL);
while (buf = getnext(read_sysencrkeys))
{
 /* Copy the current row so that fields can be updated */
 (void)copyrow((int) SYSENCRYPTKEYS,
   (BYTE *)read_sysencrkeys->srow, lencol,
   (BYTE *)&encryptkey);
 if (!(encryptkey.ekstatus & EK_SYSENCRPASS))
 {
  /*
  ** In future releases this will mean key is
  ** encrypted with a user key.
  */
  continue;
 }
 if (newplen == 0)
 {
  /*
  ** Disallow unsetting the system encryption
  ** password if one or more keys are dependent
  ** on it.
  */
  ex_raise(ENCR_SYSPASSWD_DEPEND, EX_USER, 1);
 }
 saltp = (BYTE *)&encryptkey.ekpasswd[0];
 /* Save off the version */
 MEMMOVE(&encryptkey.ekpasswd[0], &vers, ENCR_VERSION_LEN);
 /*
 ** Value of ekpasswd consists of 2 bytes of version
 ** and 8 bytes of "salt".
 */
 saltp += ENCR_VERSION_LEN;
 /*
 ** Set up static keys for decrypting and re-encrypting keys.
 ** These static keys are based on the old/new passwords.
 ** First, make key with old password for decrypting key.
 */
 okeklen = EN_AES_DIGEST_LEN;
 if (!encr_make_static_key(EK_UKEY, vers, (BYTE *)oldpasswd,
  oldplen, saltp, ENCR_SALT_LEN, &okekbuf[0],
  &okeklen))
 {
  ex_raise(ENCR_STATIC_KEY, EX_INTOK, 1);
 }
 SYB_ASSERT(okeklen == EN_AES_DIGEST_LEN);
 /*
 ** Then make static key with new password for
 ** re-encrypting key.  Use the current version of
 ** encryption algorithm for static key.
 */
 nkeklen = EN_AES_DIGEST_LEN;
 if (!encr_make_static_key(EK_UKEY, EK_STATIC_VERS,
  (BYTE *)newpasswd, newplen, saltp,
  ENCR_SALT_LEN, &nkekbuf[0], &nkeklen))
 {
```

```
 /* Error already reported */
 goto cleanup;
}
SYB_ASSERT(nkeklen == EN_AES_DIGEST_LEN);
/*
** Decrypt and re-encrypt key read from
** sysencryptkeys.  Use the same salt.
*/
if (!encr_decrypt_key_n_salt(&okekbuf[0], saltp,
 &encryptkey.ekvalue[0], encryptkey.eklen,
 EN_AES_KEY_BUFLEN, &plainbuf[0]))
{
 /* Error already reported */
 goto cleanup;
}
if (!encr_encrypt_key_n_salt(&nkekbuf[0], saltp,
 &plainbuf[0], encryptkey.eklen,
 EK_MAX_SYMKEY_VALUE_LEN, &encryptkey.ekvalue[0]))
{
 /* Error already reported */
 goto cleanup;
}
rowlen = fmtrow((int)SYSENCRYPTKEYS, (BYTE *)&encryptkey,
  lencol, databuf);
if ((xact_beginupdate(xdes, write_sysencrkeys, XMOD_DIRECT,
 0)) != XACTRV_SUCCESS)
{
 /* Error reported */
 goto cleanup;
}
copy.xdes = xdes;
SDES_CLONE_SROW(read_sysencrkeys, write_sysencrkeys);
if ((!update(write_sysencrkeys, databuf, rowlen, buf))
    || (xact_endupdate(xdes) != XACTRV_SUCCESS))
{
 /* Error already reported */
 goto cleanup;
}
/*
** Update schema count in sysobjects for the changed
** key, so that stored procedures will be recompiled
*/
initarg(mod_sysobjects, keys1, 1);
setarg(mod_sysobjects, &Sysobjects[OBJ_ID], EQ,
 (BYTE *)&encryptkey.encrkeyid, sizeof(objid_t));
startscan(mod_sysobjects, SCAN_CLUST, SCAN_FIRST);
/* Share the rowbuf pointer */
if (buf = getnext(mod_sysobjects))
{
   objrow = (OBJECT *)mod_sysobjects->srow;
   objschema2 = GETSHORT(&objrow->objostat.objschema2);
   objschema2++;
   if (!modify_row(xdes, mod_sysobjects, buf, (BYTE *)NULL,
    OFFSETOF(OBJECT, objostat.objschema2),
    (BYTE *)&objschema2, sizeof(int16),
    XREC_MODIFY, 0, TRUE))
   {
    ex_raise(SYSTEM, SYS_XACTABORT,EX_CONTROL, 0);
   }
  }
 }
 copy.retstat = SUCCEED;
cleanup:
 if (copy.er_sdes)
 {
  CLOSE_SDES(&copy.er_sdes);
 }
 if (copy.ew_sdes)
```

```
 {
  CLOSE_SDES(&copy.ew_sdes);
 }
 if (copy.o_sdes)
 {
  CLOSE_SDES(&copy.o_sdes);
 }
 if (copy.xdes)
 {
  (void)xact_endupdate(xdes);
 }
 return copy.retstat;
#else
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 5);
 return FALSE;
#endif /* USE_SECURITYBUILDER */
}
/*
** EA_ENCRYPT_SYSPASSWD
**
** Description:
** Given a plaintext password, encrypt it, along with its length
** and some validation salt, using an internal key.  Return the
** encrypted value as a hex string, appended with the hex salt
** and a sentinal byte.
**
** The salt is stored within and without the encrypted string
** so that, upon decryption, the static key generated by ASE can
** be validated as the correct key.
**
** Parameters:
** ktype  (In) Type of static key
** vers  (In) Version of static key
** plainpasswd (In) Raw system encryption passwd
** passwdlen (In) Length of password
** hex_encr_passwd (Out)Buffer for hex version of encrypted password
** hexlen  (In/Out) Length of buffer/outgoing password
**
** Returns:
** SUCCEED - Password successfully encrypted
** FAIL    - (Caller handles error)
**
*/
int
ea_encrypt_syspasswd(int ktype, int vers, char *plainpasswd, int passwdl
en,
        char *hex_encr_passwd, int *hexlen)
{
#if USE_SECURITYBUILDER
EN_GLOBALCTX *encrGctx; /* Global encryption context */
EN_LOCALCTX *encrLctx; /* Local encryption context */
BYTE  len;  /* For storing with passwd */
int  rounded_len; /* passwd+len rounded up */
BYTE  keystatic[EN_AES_DIGEST_LEN]; /* To mix key */
size_t  kstaticlen; /* Length of static key bytes */
char  plainbuf[EAMAXENCRBUF+1]; /* Passwd+len rounded up */
BYTE  cipherbuf[EAMAXENCRBUF]; /* Encrypted password */
BYTE  salt[ENCR_SALT_LEN]; /* Generated random salt */
char  hex_salt[EASALTHEXLEN+1]; /* Salt, converted to hex */
char  *errdesc; /* Pointer to error phrases */
int  retstat;
/* Initialize */
retstat = FAIL;
/* Gather context */
encrGctx = Kernel->kencr_ctx;
if (!(encrLctx = (EN_LOCALCTX *)ubfalloc(Kernel->kencr_mempool,
  sizeof(EN_LOCALCTX))))
{
```

```
 ex_callprint(EX_NUMBER(
  ENCRYPTION, ENCR_NOMEMORY), EX_RESOURCE, 2);
 ex_raise(ENCRYPTION, ENCR_NOMEMORY, EX_CONTROL, 0);
}
MEMZERO((BYTE *)encrLctx, sizeof(EN_LOCALCTX));
/* We encrypt 1-byte length along with password */
len = (BYTE)passwdlen;
/*
** Derive length from lengths of
**  - 1-byte len field
**  - passwd
**  - 8 bytes binary salt in hex format => 16 bytes,
** all rounded up to a size that is a multiple of block size.
*/
rounded_len = ((((int)len+EASALTHEXLEN)/EN_AES_BLOCKSIZE) +1)
  * EN_AES_BLOCKSIZE;
/*
** Sanity check that caller's buf accommodates twice the size
** of the rounded-up password, plus room for '0x', concatenated
** salt, sentinel byte and null terminator.
*/
SYB_ASSERT(*hexlen >= (rounded_len*2 + EASALTHEXLEN + 4));
/* Generate salt for static key creation */
if ((en_generateRandomData(encrGctx, (BYTE *)&salt[0],
   (size_t)ENCR_SALT_LEN)) != SUCCEED)
{
 ex_callprint(EX_NUMBER(
  ENCRYPTION,ENCR_BAD_RANDOM_GEN), EX_INTOK, 3);
 return FAIL;
}
/* Convert salt to hex format */
(void)bintostr(&salt[0], ENCR_SALT_LEN, (char *)&hex_salt[0]);
hex_salt[EASALTHEXLEN] = '\0';
/*
** Concatenate all parts ready for encryption.  Move in password
** length followed by password and hex salt.
*/
plainbuf[0] = (BYTE)len;
strncpy(&plainbuf[1], plainpasswd, passwdlen);
strncpy(&plainbuf[1+passwdlen], hex_salt, EASALTHEXLEN);
kstaticlen = EN_AES_DIGEST_LEN;
/* Mix salt and static bytes for static encryption key */
if (!encr_make_static_key(ktype, vers, &salt[0], ENCR_SALT_LEN,
   NULL, 0, &keystatic[0], &kstaticlen))
{
 goto fail;
}
/* Create encryption key from internal static data */
if (!en_aes_beginCryptOper(encrGctx, encrLctx, &keystatic[0],
  EN_SYSTEM_BIT_KEYSIZE, EN_ENCRYPT, NULL, 0,
  &errdesc))
{
 goto fail;
}
if (!en_aes_encrypt(encrGctx, encrLctx, (BYTE *)&plainbuf[0],
  rounded_len, &cipherbuf[0], &errdesc))
{
 goto fail;
}
if (!en_aes_endCryptOper(encrGctx, encrLctx, &errdesc))
{
 goto fail;
}
/* Convert encrypted password to hex in caller's buffer */
hex_encr_passwd[0] = '0';
hex_encr_passwd[1] = 'x';
*hexlen = 2;
*hexlen += bintostr(&cipherbuf[0], rounded_len, &hex_encr_passwd[2]);
```

```
 /* Concatenate salt, for later validation of decryption */
 strncpy(&hex_encr_passwd[*hexlen], hex_salt, EASALTHEXLEN);
 *hexlen += EASALTHEXLEN;
 /* Concatenate sentinel byte containing char '1' */
 hex_encr_passwd[(*hexlen)++] = '1';
 hex_encr_passwd[*hexlen] = '0';
 SYB_ASSERT(*hexlen == (rounded_len*2 + EASALTHEXLEN + 3));
 retstat = SUCCEED;
 /* Fall through for clean up */
fail:
 MEMZERO((BYTE *)&plainbuf[0], EAMAXENCRBUF);
 if (encrLctx)
 {
  ubffree(Kernel->kencr_mempool, (void *)encrLctx);
 }
 return retstat;
#else /* USE_SECURITYBUILDER */
 ex_callprint(EX_NUMBER(ENCRYPTION, ENCR_PLATFORM), EX_INTOK, 1);
 return FAIL;
#endif /* USE_SECURITYBUILDER */
}
/*
** EA_DECRYPT_SYSPASSWD
**
** Description:
** Given a buffer containing a hex string representation of
** an encrypted password, appended with some hex salt and a
** sentinel byte, return a plaintext password and its length.
** If the appended salt doesn't match the encrypted salt, it
** could indicate an inconsistency between the version of the
** static key used to encrypt the password and the version of
** the static key used to decrypt the password. It could also
** indicate a corruption of sysencryptkeys.
**
** Note: the hex string passed in is a hex translation of
** - 1 byte len
** - password + hex salt, rounded up to blocklength
** - appended hex salt
** - sentinel byte
**
** Parameters:
** ktype  (In) Type of static key
** vers  (In) Version of static key
** hex_encr_passwd (In) Encrypted passwd in hex
** hexlen  (In) Length of encrypted password
** plainpasswd (Out)Buffer for plain password
** passlen  (In/Out) Length of buffer/outgoing password
**
** Returns:
** SUCCEED - Password successfully decrypted
** FAIL    - (Caller handles error)
**
*/
int
ea_decrypt_syspasswd(int ktype, int vers, char *hex_encr_passwd, int hex
len,
   BYTE *plainpasswd, int *passlen)
{
#if USE_SECURITYBUILDER
 EN_GLOBALCTX *encrGctx; /* Global encryption context */
 EN_LOCALCTX *encrLctx; /* Local encryption context */
 int  len;  /* Intermediate length */
 BYTE  keystatic[EN_AES_DIGEST_LEN]; /* To mix key */
 size_t  kstaticlen; /* Length of static key bytes */
 int  cipherlen; /* Length of encrypted passwd */
 BYTE  cipherbuf[EAMAXENCRBUF]; /* Encrypted password */
 char   plainbuf[EAMAXENCRBUF]; /* Password and len */
 BYTE  salt[ENCR_SALT_LEN]; /* Binary salt */
```

```
char  *hex_saltp; /* Pointer to salt of hex input */
char  *errdesc; /* For return error codes */
int  retstat;
retstat = FAIL;
/* Get ASE context structures for decryption */
encrGctx = Kernel->kencr_ctx;
if (!(encrLctx = ubfalloc(Kernel->kencr_mempool, sizeof(EN_LOCALCTX))))
{
 ex_callprint(EX_NUMBER(
  ENCRYPTION, ENCR_NOMEMORY), EX_RESOURCE, 1);
 ex_raise(ENCRYPTION, ENCR_NOMEMORY, EX_CONTROL, 0);
}
MEMZERO((BYTE *)encrLctx, sizeof(EN_LOCALCTX));
/*
** Strip off appended salt and sentinel byte. Convert salt to
** binary
*/
hexlen -= (EASALTHEXLEN +1);
hex_saltp = &hex_encr_passwd[hexlen];
if ((len = strtobin(hex_saltp, EASALTHEXLEN, &salt[0])) == 0)
{
 ex_callprint(
     EX_NUMBER(ENCRYPTION,ENCR_SYSPASS_CORRUPT), EX_INTOK, 1);
 goto fail;
}
SYB_ASSERT(len == ENCR_SALT_LEN);
/* Convert the encrypted passwd/salt from hex to binary */
if ((len = strtobin(hex_encr_passwd, hexlen, &cipherbuf[0])) == 0)
{
 ex_callprint(
     EX_NUMBER(ENCRYPTION,ENCR_SYSPASS_CORRUPT), EX_INTOK, 2);
 goto fail;
}
SYB_ASSERT(len <= EAMAXENCRBUF);
if (len % EN_AES_BLOCKSIZE != 0)
{
 ex_callprint(
     EX_NUMBER(ENCRYPTION,ENCR_SYSPASS_CORRUPT), EX_INTOK, 3);
 goto fail;
}
/* Mix salt and static bytes for static encryption key */
kstaticlen = EN_AES_DIGEST_LEN;
if (!encr_make_static_key(ktype, vers, &salt[0], ENCR_SALT_LEN, NULL, 0
,
   &keystatic[0], &kstaticlen))
{
 ex_callprint(EX_NUMBER(
  ENCRYPTION,ENCR_STATIC_KEY), EX_INTOK, 2);
 goto fail;
}
/* Create decryption key from internal static data */
if (!en_aes_beginCryptOper(encrGctx, encrLctx, &keystatic[0],
  EN_SYSTEM_BIT_KEYSIZE, EN_DECRYPT,
  NULL, 0, &errdesc))
{
 ex_callprint(EX_NUMBER(
  ENCRYPTION,ENCR_SETUP_FAIL), EX_INTOK, 6);
 goto fail;
}
if (!(en_aes_decrypt(encrGctx, encrLctx, &cipherbuf[0], len,
  (BYTE *)&plainbuf[0], &errdesc)))
{
 ex_callprint(EX_NUMBER(
  ENCRYPTION,ENCR_DECRYPTION_FAIL), EX_INTOK, 4);
 goto fail;
}
if (!en_aes_endCryptOper(encrGctx, encrLctx, &errdesc))
{
```

```
 ex_callprint(EX_NUMBER(
  ENCRYPTION,ENCR_END_FAIL), EX_INTOK, 5);
 goto fail;
}
/* Extract length and password */
SYB_ASSERT(*passlen > (BYTE)plainbuf[0]);
*passlen = (BYTE)plainbuf[0];
/*
** Point at decrypted salt (it's hex) and compare to appended
** salt in hex_encr_passwd.
*/
if (STRNCMP(&plainbuf[*passlen+1], hex_saltp, EASALTHEXLEN))
{
 /*
 ** If the salt appended to the sysattributes row
 ** doesn't match the salt that was encrypted with
 ** the password, we can't rely on the decrypted
 ** system password.
 */
 ex_callprint(EX_NUMBER(
  ENCRYPTION,ENCR_SYSPASSWD_DECRYPT), EX_USER, 1);
  goto fail;
 }
 strncpy((char *)plainpasswd, &plainbuf[1], *passlen);
 retstat = SUCCEED;
fail:
 MEMZERO((BYTE *)&plainbuf[0], EAMAXENCRBUF);
 if (encrLctx)
 {
  ubffree(Kernel->kencr_mempool, (void *)encrLctx);
 }
 return retstat;
#else /* USE_SECURITYBUILDER */
 ex_callprint(EX_NUMBER(ENCRYPTION, ENCR_PLATFORM), EX_INTOK, 2);
 return FAIL;
#endif /* USE_SECURITYBUILDER */
}
/*
** ENCR_ADMIN
**
** Description:
** This function implements the encr_admin builtin function, which
** does the ASE internal work of the sp_encryption stored procedure.
**
** Parameters:
** const1  Constant for the command passed from sp_encryption.
**   For the moment the command could be:
**    'help'
**    'system_encr_passwd'
**
** const2  Constant for second parameter: stores information
**   about the new system password.
**
** const3  Constant for third parameter: stores the information
**   related to the old system password.
**
** const4  Constant for forth parameter: format
** const5  Constant for fifth parameter: version
**
** Returns:
** 0 SUCCESS  values useful in sproc that calls this function.
** 1 FAIL
**
** Side Effects:
** After the call to ea__setpasswd(), const2 and const3 would be filled
** with the cyphered password, encrypted for replication.
**
** Assumptions:
```

```
**
*/
#if USE_SECURITYBUILDER
int32
encr_admin(CONSTANT *const1, CONSTANT *const2, CONSTANT *const3,
  CONSTANT *const4, CONSTANT *const5)
{
 char *cmd;  /* Command passed from sp_encryption  */
 char *arg2;  /* Second parameter in sp_encryption */
 char *arg3;  /* Third parameter in sp_encryption */
 int *fmt;  /* Format of passwords   */
 int *vers;  /* Version of static mixing algorithm */
 int     len1;  /* Command's length passed to sp_encryption  */
 CS_INT *plen2;  /* Ptr to the new passwd length in the const */
 CS_INT *plen3;  /* Ptr to the old passwd length in the const */
 CS_INT maxarg2_len; /* Max length of the buffer holding the new
    ** password.          */
 CS_INT maxarg3_len; /* Max length of the buffer holding the old
    ** password.          */
 int i;
 int retval = FALSE;
int encr_oper;
/* Init local variables */
cmd = (char *) const1->value;
arg2 = (char *) const2->value;
arg3 = (char *) const3->value;
len1 = const1->len;
plen2 = &(const2->len);
plen3 = &(const3->len);
maxarg2_len = const2->maxlen;
maxarg3_len = const3->maxlen;
/* is the old passwd missing ? */
(*plen3) = BI_MISSING_PARM(TRUE, const3) ? 0 : (const3->len);
/* Have been set fmt and vers parameters ? */
fmt = (BI_MISSING_PARM(TRUE, const4) || ((int) const4->len == 0))
  ? NULL : (int *) const4->value;
vers = (BI_MISSING_PARM(TRUE, const5) || ((int) const5->len == 0))
  ? NULL : (int *) const5->value;
if (!(CFG_GETCURVAL(cfgencryptedcols)))
{
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_NO_CONFIG), EX_USER,1,
   "sp_encryption");
 goto errorout;
}
/* A command has to be passed to sp_encryption  */
if ((len1 <= 0) || (cmd == NULL))
{
 ex_callprint(EX_NUMBER(
  ENCRYPTION,ENCR_MISSING_PARAM), EX_USER, 1,
   "<command>");
 goto errorout;
}
/* Match command */
if ((len1 >= 18) && (STRNCMP(cmd, "system_encr_passwd", 18) == 0))
{
 encr_oper = EN_SET_PASSWD;
}
/*
** ENCRCOLS_RESOLVE: Other sp_encryption operations, when
** implemented, should be checked here
** else if (STRNCMP(cmd, "help", 4) == 0)
*/
else
{
 ex_callprint(EX_NUMBER(PARSER, P_OPTION1), EX_SYNTAX, 1,
   len1, cmd, PH_PARAM);
 goto errorout;
}
```

```
switch (encr_oper) {
  case EN_SET_PASSWD:
 if ((*plen2) == 0 && (*plen3) == 0)
 {
  ex_callprint(EX_NUMBER(
   ENCRYPTION,ENCR_MISSING_PARAM), EX_USER, 1,
    "<newpasswd>");
  goto errorout;
 }
 /* ENCRCOLS_RESOLVE: Reviewers, do we need to send in minor
 ** arguments - fmt and version? */
 if ((rvm_bi_encr_admin(cmd, len1, arg2, (*plen2),
       arg3, (*plen3)))
     != RVM_OK)
 {
  /* Permissions-related error already printed */
  goto errorout;
 }
 /* Encrypt the password */
 retval = ea__setpasswd(arg2, plen2, arg3, plen3,
    maxarg2_len, maxarg3_len,
      fmt, vers);
errorout:
  /* Convert TRUE/FALSE to SQL/sproc SUCCESS (0) or FAIL (1) */
  if (retval)
  {
   /* Replication needs two special parameters. */
   /* Set format parameter */
   (*((int *)(const4->value))) = (int)EK_HEX_SYSENCRPASSWD;
   (const4->len) =  sizeof(int);
   /* Set static encryption version */
   (*((int *)(const5->value))) = (int)EK_STATIC_VERS;
   const5->len = sizeof(int);
   return (0); /* SUCCESS */
  }
  else
  {
   /* Clear password parameters from memory */
   MEMZERO(arg2, (*plen2));
   MEMZERO(arg3, (*plen3));
   len1 = (*plen2) = (*plen3) = -1;
   return (1);       /* FAIL */
  }
     default:
  SYB_ASSERT(0);
 }
}
#else
/* stub for platforms that do not support encrypted columns */
int32
encr_admin( CONSTANT *const1, CONSTANT *const2, CONSTANT *const3,
  CONSTANT *const4, CONSTANT *const5)
{
 ex_callprint(EX_NUMBER(ENCRYPTION,ENCR_PLATFORM), EX_INTOK, 4);
 return -1;
}
#endif /* USE_SECURITYBUILDER */
// encryption.c
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
/*
** Copyright (C) 2003 Sybase, Inc.
**
** Sybase, Inc.  All rights reserved.
** Unpublished rights reserved under U.S. copyright laws.
**
** This software contains confidential and trade secret information of S
ybase,
** Inc.   Use,  duplication or disclosure of the software and documentat
```

```
ion by
** the  U.S.  Government  is  subject  to  restrictions set forth in a l
icense
** agreement  between  the  Government  and  Sybase,  Inc.  or  other  w
ritten
** agreement  specifying  the  Government's rights to use the software a
nd any
** applicable FAR provisions, for example, FAR 52.227-19.
** Sybase, Inc. One Sybase Drive, Dublin, CA 94568, USA
*/
/*
** File: encryption.c
**
** The functions defined here use the Certicom Security Builder API.
** The file is conditionally compiled. For non-SB platforms, function
** stubs are defined at the end of the file to satisfy the linker.
*/
/*
** ASE headers
*/
#include <port.h> /* always required as first sybase include file */
#include <syb_std.h> /* always required as second sybase include file */
#include <dtypes.h> /* always required as third sybase include file */
#include <server.h> /* always required as fourth sybase include file */
#include <bitbyte.h> /* required for TRACEPRINT */
#include <trace.h> /* required for TRACEPRINT */
#include <derror.h> /* required for TRACEPRINT */
#include <uksrc_dcl.h> /* required for upyield() */
#include <cachemgr.h>
#include <kernel.h>
#include <memfrg.h>
#include <datetime.h>
#include <tod.h> /* required for utget()  */
#include <encryption.h>
#if USE_SECURITYBUILDER
/*
** Security Builder headers
*/
#if USE_SB4
#include <sbctx.h>
#endif /* USE_SB4 */
#include <sbaes.h>
#include <sbsha1.h>
#include <sbyield.h>
#include <sbreturn.h>
#include <sbrandom.h>
#include <string.h>
/*******************DEFINE AS 1 FOR DEBUGGING*****************************
/
#define STATIC_KEY_DEBUG 1
/*****************************DEBUGGING END*********************************
/
/* Size of seed for random number generation */
#define SEEDSIZE_BYTES 32
#define IV_LEN  SB_AES_128_BLOCK_BITS/BITS_IN(BYTE)
#define TRACESB  if (TRACECMDLINE(ENCRYPTION,1)) TRACEPRINT
/*
** ENCRCOLS_RESOLVE: May be a good idea to make this much bigger
** so that reseeding will not be so frequent
*/
static BYTE seedValue[SEEDSIZE_BYTES];
/* Forward references */
SYB_STATIC int en_seed(void *, size_t, BYTE *, void *);
SYB_STATIC char *en__geterr(int);
int en_yieldCallback(void *);
void *en_malloc(size_t, void *);
void en_memcpy(void *, const void *, size_t, void *);
int en_memcmp(const void *, const void *, size_t, void *);
```

```
void en_memset(void *, int, size_t, void *);
void en_free(void *, void *);
#if STATIC_KEY_DEBUG
SYB_STATIC void
en_mkhexstr(char **outbp, BYTE *datap, int datlen);
#endif /* STATIC_KEY_DEBUG */
/*
** Error condition lookup table, for use in printing error messages
*/
typedef struct {
 int ecode;
 char *edesc;
} EN_ERRS;
static EN_ERRS Err_lookup[] =
{
 {SB_FAILURE,    "SB_FAILURE"},
 {SB_NOT_IMPLEMENTED,   "SB_NOT_IMPLEMENTED"},
 {SB_ERR_NULL_PARAMS,   "SB_ERR_NULL_PARAMS"},
 {SB_ERR_NULL_PARAMS_PTR, "SB_ERR_NULL_PARAMS_PTR"},
 {SB_ERR_BAD_MODE,   "SB_ERR_BAD_MODE"},
 {SB_ERR_BAD_BLOCK_LEN,   "SB_ERR_BAD_BLOCK_LEN"},
 {SB_ERR_BAD_PARAMS,   "SB_ERR_BAD_PARAMS"},
#if USE_SB4
 {SB_ERR_BAD_SB_CONTEXT,   "SB_ERR_BAD_SB_CONTEXT"},
#endif /* USE_SB4 */
 {SB_FAIL_ALLOC,    "SB_FAIL_ALLOC"},
 {SB_ERR_NO_RNG,    "SB_ERR_NO_RNG"},
 {SB_ERR_BAD_KEY,   "SB_ERR_BAD_KEY"},
 {SB_ERR_BAD_KEY_LEN,   "SB_ERR_BAD_KEY_LEN"},
 {SB_ERR_NULL_KEY_PTR,   "SB_ERR_NULL_KEY_PTR"},
 {SB_ERR_NULL_CONTEXT,   "SB_ERR_NULL_CONTEXT"},
 {SB_ERR_NULL_CONTEXT_PTR, "SB_ERR_NULL_CONTEXT_PTR"},
 {SB_ERR_BAD_CONTEXT,   "SB_ERR_BAD_CONTEXT"},
 {SB_ERR_NULL_KEY,   "SB_ERR_NULL_KEY"},
 {SB_ERR_NULL_INPUT_BUF,   "SB_ERR_NULL_INPUT_BUF"},
 {SB_ERR_BAD_INPUT_BUF_LEN, "SB_ERR_BAD_INPUT_BUF_LEN"},
 {SB_ERR_NULL_OUTPUT_BUF, "SB_ERR_NULL_OUTPUT_BUF"},
 {SB_ERR_BAD_OUTPUT_BUF_LEN, "SB_ERR_BAD_OUTPUT_BUF_LEN"},
 {SB_ERR_NULL_IV,   "SB_ERR_NULL_IV"},
 {SB_ERR_BAD_IV_LEN,   "SB_ERR_BAD_IV_LEN"},
 {0,     NULL}
};
/*
**  EN_SHA1_DIGEST
**
**  Security Builder wrapper function to make a digest from a string.
**
**  Parameters:
** encrGctx - (in) pointer global SB context structure
** message -  (in) Pointer to message to be 'digested'
** msglen  -  (in) Length of message
** digest -  (in) Pointer to buffer for digest result
** diglen  -  (in/out) Length of buffer/length of digest
** errdesc -  (out) Address of pointer to SB diagnostic msg
**
**  Returns:
** SUCCEED/FAIL
**
*/
int
en_sha1_digest(EN_GLOBALCTX * encrGctx, BYTE *message, size_t msglen,
  BYTE *digest, size_t *diglen, char **errdesc)
{
 sb_YieldCtx yieldctx;
 sb_Context  sha1Context;
 void  *sbctx;
 int  returncode;
 sbctx = encrGctx->en_sbgctx;
```

```
yieldctx = (sb_YieldCtx)encrGctx->en_gyieldctx;
sha1Context = NULL;
returncode = SB_SUCCESS;
*errdesc = NULL;
SYB_ASSERT(*diglen >= SB_SHA1_DIGEST_LEN);
/* Initialize SHA-1 Context */
returncode = sb_SHA1Begin((size_t)SB_SHA1_DIGEST_LEN,
    yieldctx, &sha1Context, sbctx);
if (returncode != SB_SUCCESS)
{
 TRACESB("sb_SHA1Begin failed with error: %s\n",
  en__geterr(returncode));
 goto fail;
}
/* Hash message */
returncode = sb_SHA1Hash(sha1Context, msglen, message, sbctx);
if (returncode != SB_SUCCESS)
{
 TRACESB("sb_SHA1Hash failed with error: %s\n",
  en__geterr(returncode));
 goto fail;
 }
 /* Complete hashing */
 returncode = sb_SHA1End(&sha1Context, digest, sbctx);
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_SHA1End failed with error: %s\n",
   en__geterr(returncode));
  goto fail;
 }
 *diglen = SB_SHA1_DIGEST_LEN;
 return SUCCEED;
fail:
 if (returncode != SB_SUCCESS)
 {
  *errdesc = en__geterr(returncode);
 }
 return FAIL;
}
/*
**  EN_AES_CREATESYMKEY
**
**  Security Builder wrapper function to create a symmetric key
**  for the CREATE ENCRYPTION KEY command.
**
**  Parameters:
** encrGctx - (in) pointer global SB context structure
** keysize - (in) Requested key size in bits
** status  - (in) Key/encryption attributes
** keybuf - (out) Pointer to buffer for binary key
** kbuflen - (in/out) Length of keybuf/length of binary key in bytes
** errdesc - (out) Address of pointer to SB diagnostic msg
**
**  Returns:
** SUCCEED/FAIL
**
*/
int
en_aes_createsymkey(EN_GLOBALCTX *encrGctx, size_t keysize, int status,
 BYTE *keybuf, size_t *kbuflen, char **errdesc)
{
 int  returncode;
 sb_YieldCtx  yieldctx;
 void  *sbctx;
 sb_RNGCtx rngctx;
 sb_Params encrParams;
 sb_Key  encrKey;
 int  mode;
```

```
size_t  keybitlen;
returncode = SB_SUCCESS;
yieldctx = (sb_YieldCtx)encrGctx->en_gyieldctx;
sbctx = encrGctx->en_sbgctx;
rngctx = encrGctx->en_grngctx;
errdesc = NULL;
encrKey = NULL;
keybitlen = *kbuflen * BITS_IN(BYTE);
/*
** Request for use of init vector implies encryption using
** Cipher Block Chaining mode; otherwise Electronic Code Book mode.
** This is a Security Builder restriction.
*/
mode = (status & EN_INIT_VECTOR) ? SB_AES_CBC : SB_AES_ECB;
/*
** Set up AES parameters for key. Up through SB4 only available
** block length is 128
*/
returncode = sb_AESParamsCreate(mode, SB_AES_128_BLOCK_BITS,
  rngctx, yieldctx, &encrParams, sbctx);
if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_AESParamsCreate failed with error: %s\n",
   en__geterr(returncode));
  goto fail;
 }
 /*
 ** Generate encryption key
 */
#if USE_SB4
 returncode = sb_AESEncryptKeyCreate(encrParams,
  keysize, NULL, &encrKey, sbctx);
#else
 returncode = sb_AESKeyCreate(encrParams,
  keysize, NULL, &encrKey, sbctx);
#endif /* USE_SB4 */
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_AESEncryptKeyCreate or sb_AESKeyCreate failed with error:
%s\n",
   en__geterr(returncode));
  goto fail;
 }
 /*
 ** Transform SB key object into a binary string
 */
 returncode = sb_AESKeyGet(encrParams, encrKey, &keybitlen,
   keybuf, sbctx);
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_AESKeyGet failed with error: %s\n",
   en__geterr(returncode));
  *kbuflen = 0;
  goto fail;
 }
 *kbuflen = keybitlen/BITS_IN(BYTE);
 /* Fall through to destroy sb_Key and params structures */
fail:
 if (returncode != SB_SUCCESS)
 {
  *errdesc = en__geterr(returncode);
 }
 returncode = sb_AESKeyDestroy(encrParams, &encrKey, sbctx);
 if (returncode != SB_SUCCESS)
 {
  if (!*errdesc)
  {
   TRACESB("sb_AESKeyDestroy failed with error: %s\n",
```

```
   en__geterr(returncode));
  *errdesc = en__geterr(returncode);
 }
}
returncode = sb_AESParamsDestroy(&encrParams, sbctx);
if (returncode != SB_SUCCESS)
{
 if (!*errdesc)
 {
  TRACESB("sb_AESParamsDestroy failed with error: %s\n",
   en__geterr(returncode));
  *errdesc = en__geterr(returncode);
 }
}
if (returncode != SB_SUCCESS)
{
 *kbuflen = 0;
 return FAIL;
}
else
{
  return SUCCEED;
 }
}
/*
**  EN_AES_BEGINCRYPTOPER
**
**  Security Builder wrapper function to set up for a symmetric
**  encryption or decryption operation.
**
**  Parameters:
** encrGctx - (in) pointer to global SB context structure
** encrLctx - (in/out) pointer to empty local context structure,
**     filled in by this function
** key      - (in) Encryption key as binary string
** keysize  - (in) Size (in bits) of symmetric key
** status   - (in) Key/encryption attributes
** ivbuf - (out) Pointer to buffer containing init vector
** ivlen -   (in) Length of iv
** errdesc - (out) Address of pointer to SB diagnostic msg
**
**  Returns:
** SUCCEED/FAIL
**
*/
int
en_aes_beginCryptOper(EN_GLOBALCTX *encrGctx, EN_LOCALCTX *encrLctx,
  unsigned char *key, size_t keysize, int status,
  unsigned char *ivbuf, int ivlen, char **errdesc)
{
 int  returncode;
 void  *sbctx;
 sb_YieldCtx  sbyieldctx;
 sb_Params sbencrparams;
 sb_Context sbencrctx;
 sb_Key  sbencrkey;
 int  mode;
 TRACESB("ENTER: en_aes_beginCryptOper(%p, %p, %p, %u, %d, %p, %d, %p)\n
",
  encrGctx,encrLctx,key,keysize,status,ivbuf,ivlen,errdesc);
 SYB_ASSERT(encrLctx && encrGctx);
 /* Initialize */
 returncode = SB_SUCCESS;
 sbencrparams = NULL;
 sbencrctx = NULL;
 sbencrkey = NULL;
 *errdesc = NULL;
 sbctx = encrGctx->en_sbgctx;
```

```
sbyieldctx = encrGctx->en_gyieldctx;
/*
** Request for use of init vector implies encryption using
** Cipher Block Chaining mode; otherwise Electronic Code Book
** mode.  This is a Security Builder restriction.
*/
mode = (status & EN_INIT_VECTOR) ? SB_AES_CBC : SB_AES_ECB;
/* Set up AES parameters for encryption or decryption. */
returncode = sb_AESParamsCreate(mode, SB_AES_128_BLOCK_BITS,
  NULL, sbyieldctx, &sbencrparams, sbctx);
if (returncode != SB_SUCCESS)
{
 TRACESB("sb_AESParamsCreate failed with error: %s\n",
   en__geterr(returncode));
 goto fail;
}
/* Instantiate AES key object from binary string */
if (status & EN_ENCRYPT)
{
 returncode = sb_AESEncryptKeyCreate(sbencrparams, keysize,
  (const unsigned char *)key, &sbencrkey, sbctx);
 }
 else
 {
  SYB_ASSERT(status & EN_DECRYPT);
  returncode = sb_AESDecryptKeyCreate(sbencrparams, keysize,
   (const unsigned char *)key, &sbencrkey, sbctx);
 }
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_AESEncryptKeyCreate or sb_AESDecryptKeyCreate failed with
error: %s\n",
    en__geterr(returncode));
  goto fail;
 }
 if (status & EN_ENCRYPT)
 {
  returncode = sb_AESEncryptBegin(sbencrparams, sbencrkey,
   ivlen, (const unsigned char *)&ivbuf[0],
   &sbencrctx, sbctx);
 }
 else /* EN_DECRYPT */
 {
  returncode = sb_AESDecryptBegin(sbencrparams, sbencrkey,
   ivlen, (const unsigned char *)&ivbuf[0],
   &sbencrctx, sbctx);
 }
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_AESEncryptBegin or sb_AESDecryptBegin failed with error: %
s\n",
    en__geterr(returncode));
  goto fail;
 }
 encrLctx->en_sblparams = (void *)sbencrparams;
 encrLctx->en_sblctx = (void *)sbencrctx;
 encrLctx->en_sblkey = (void *)sbencrkey;
 return SUCCEED;
fail:
 if (returncode != SB_SUCCESS)
 {
  *errdesc = en__geterr(returncode);
 }
 /*
 ** Destroy AES encryption/decryption context, key object and
 ** params.
 */
 if (sbencrctx)
```

```
 {
  (void)sb_AESEnd(&sbencrctx, sbctx);
 }
 if (sbencrkey)
 {
  (void)sb_AESKeyDestroy(sbencrparams, &sbencrkey, sbctx);
 }
 if (sbencrparams)
 {
  (void)sb_AESParamsDestroy(&sbencrparams, sbctx);
 }
 return FAIL;
}
/*
**  EN_AES_ENCRYPT
**
**  Security Builder wrapper function to encrypt data using AES algorith
m.
**
**  Parameters:
** encrGctx - (in) pointer to global SB context structure
** encrLctx - (in) pointer to local context structure,
** plaintext- (in) buffer of data for encryption
** plaintlen -(in) Size (in bytes) of buffer
** ciphertext - (out) Encrypted data
** errdesc - (out) Address of pointer to SB diagnostic msg
**
**  Returns:
** SUCCEED/FAIL
**
*/
int
en_aes_encrypt(EN_GLOBALCTX *encrGctx, EN_LOCALCTX *encrLctx,
  unsigned char *plaintext, size_t plaintlen,
  unsigned char *ciphertext, char **errdesc)
{
 int  returncode;
 void  *sbctx;
 sb_Context  sbencrctx;
 SYB_ASSERT(encrLctx);
 returncode = SB_SUCCESS;
 *errdesc = NULL;
 sbctx = encrGctx->en_sbgctx;
 sbencrctx = (sb_Context)encrLctx->en_sblctx;
 returncode = sb_AESEncrypt(sbencrctx, plaintlen,
   plaintext, ciphertext, sbctx);
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_AESEncrypt failed with error: %s\n",
   en__geterr(returncode));
  *errdesc = en__geterr(returncode);
  return FAIL;
 }
 return SUCCEED;
}
/*
**  EN_AES_DECRYPT
**
**  Security Builder wrapper function to decrypt data using AES algorith
m.
**
**  Parameters:
** encrGctx - (in) pointer to global SB context structure
** encrLctx - (in) pointer to local context structure,
** ciphertext-(in) Data to be decrypted
** ciphtlen - (in) Size (in bytes) of encrypted data
** plaintext- (out)Buffer of decrypted data
** errdesc - (out) Address of pointer to SB diagnostic msg
```

```
**
**  Returns:
** SUCCEED/FAIL
**
*/
int
en_aes_decrypt(EN_GLOBALCTX *encrGctx, EN_LOCALCTX *encrLctx,
  unsigned char *ciphertext, size_t ciphtlen,
  unsigned char *plaintext, char **errdesc)
{
 int  returncode;
 void  *sbctx;
 sb_Context  sbencrctx;
 SYB_ASSERT(encrLctx);
 *errdesc = NULL;
 returncode = SB_SUCCESS;
 sbctx = encrGctx->en_sbgctx;
 sbencrctx = (sb_Context)encrLctx->en_sblctx;
 returncode = sb_AESDecrypt(sbencrctx, ciphtlen,
   ciphertext, plaintext, sbctx);
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_AESDecrypt failed with error: %s\n",
   en__geterr(returncode));
  *errdesc = en__geterr(returncode);
  return FAIL;
 }
 return SUCCEED;
}
/*
**  EN_AES_ENDCRYPTOPER
**
**  Security Builder wrapper function to clean up after a symmetric
**  encryption or decryption operation.
**
**  Parameters:
** encrGctx - (in) pointer to global SB context structure
** encrLctx - (in) pointer to local context structure.
** errdesc - (out) Address of pointer to SB diagnostic msg
**
**  Returns:
** SUCCEED/FAIL
**
*/
int
en_aes_endCryptOper(EN_GLOBALCTX *encrGctx, EN_LOCALCTX *encrLctx,
  char **errdesc)
{
 int  returncode;
 void  *sbctx;
 sb_Context  sbencrctx;
 sb_Params sbencrparams;
 sb_Key  sbencrkey;
 SYB_ASSERT(encrLctx);
 *errdesc = NULL;
 returncode = SB_SUCCESS;
 sbctx = encrGctx->en_sbgctx;
 sbencrctx = (sb_Context)encrLctx->en_sblctx;
 sbencrparams = (sb_Params)encrLctx->en_sblparams;
 sbencrkey = (sb_Key)encrLctx->en_sblkey;
 returncode = sb_AESEnd(&sbencrctx, sbctx);
 encrLctx->en_sblctx = NULL;
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_AESEnd failed with error: %s\n",
   en__geterr(returncode));
  *errdesc = en__geterr(returncode);
 }
```

```
 returncode = sb_AESKeyDestroy(sbencrparams, &sbencrkey, sbctx);
 encrLctx->en_sblkey = NULL;
 if (returncode != SB_SUCCESS && !(*errdesc))
 {
  TRACESB("sb_AESKeyDestroy failed with error: %s\n",
   en__geterr(returncode));
  *errdesc = en__geterr(returncode);
 }
 returncode = sb_AESParamsDestroy(&sbencrparams, sbctx);
 encrLctx->en_sblparams = NULL;
 if (returncode != SB_SUCCESS && !(*errdesc))
 {
  TRACESB("sb_AESParamsDestroy failed with error: %s\n",
   en__geterr(returncode));
  *errdesc = en__geterr(returncode);
 }
 return (returncode == SB_SUCCESS) ? SUCCEED : FAIL;
}
/*
**  EN_INIT
**
**  Initialize global ASE encryption context by creating the
**  the Security Builder general context (Version 4 only), the yield
**  context and the random number generation context.
**
**  Parameters:
** encrGctx - (in) Pointer to global context
** errdesc  - (out) Address of pointer to SB diagnostic msg
**
**  Returns:
** SUCCEED/FAIL
**
*/
int
en_init(EN_GLOBALCTX *encrGctx, char **errdesc)
{
 int  returncode;
 void  *sbctx;
 sb_YieldCtx  yieldctx;
 sb_RNGCtx  rngctx;
 returncode = SB_SUCCESS;
 *errdesc = NULL;
 sbctx = NULL;
 yieldctx = NULL;
 rngctx = NULL;
 encrGctx->en_sbgctx = NULL;
 encrGctx->en_gyieldctx = NULL;
 encrGctx->en_grngctx = NULL;
#if USE_SB4
 /*
 ** Only SB4 has context creation and initialization
 */
 returncode = sb_SBContextCreate(
  en_malloc, en_free, NULL, NULL,
  en_memcpy, en_memcmp, en_memset,
  NULL, &sbctx);
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_SBContextCreate failed with error: %s\n",
   en__geterr(returncode));
  goto fail;
 }
#else
 /*
 ** ENCRCOLS_RESOLVE: Need to set memory func pointers here.
 ** But not until we have figured out how this will affect
 ** memory allocators set up by SSL (handled through
 ** EZ->Security Builder code)
```

```
 */
#endif /* USE_SB4 */
 encrGctx->en_sbgctx = sbctx;
 /*
 ** Create yield Context
 */
 returncode = sb_YieldCreate(en_yieldCallback, NULL,
     &yieldctx, sbctx);
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_SBYieldCreate failed with error: %s\n",
   en__geterr(returncode));
  goto fail;
 }
 encrGctx->en_gyieldctx = (void *)yieldctx;
 /* Seed the random number generation */
 returncode =  en_seed(NULL, SEEDSIZE_BYTES, (BYTE *)&seedValue[0], NULL
);
 if (returncode != SB_SUCCESS)
 {
  goto fail;
 }
#if USE_SB4
 returncode = sb_FIPS140ANSIRngCreate(SEEDSIZE_BYTES,
  (BYTE *)&seedValue[0], en_seed, NULL, yieldctx, &rngctx, sbctx);
#else /* USE_SB4 */
 returncode = sb_ANSIRngCreate(SEEDSIZE_BYTES,
  &seedValue[0], en_seed, NULL, yieldctx, &rngctx, sbctx);
#endif /* USE_SB4 */
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_FIPS140ANSIRngCreate or sb_ANSIRngCreate failed with error
: %s\n",
   en__geterr(returncode));
  goto fail;
 }
 encrGctx->en_grngctx = (void *)rngctx;
 return SUCCEED;
fail:
 if (rngctx)
 {
  (void)sb_ANSIRngDestroy(&rngctx, sbctx);
 }
 if (yieldctx)
 {
  (void)sb_YieldDestroy(&yieldctx, sbctx);
 }
#if USE_SB4
 if (sbctx)
 {
  (void)sb_SBContextDestroy(&sbctx);
 }
#endif /* USE_SB4 */
 encrGctx->en_sbgctx = NULL;
 encrGctx->en_gyieldctx = NULL;
 encrGctx->en_grngctx = NULL;
 *errdesc = en__geterr(returncode);
 return FAIL;
}
/*
** EN_CLEANUP
**
** Called when server is being shut down. This function destroys
** the global security context.
** ENCRCOLS_RESOLVE: Is there any point in this?
**
** Parameters:
** encrGctx - (in) pointer global SB context structure
```

```
**
** Returns:
** Nothing
*/
void
en_cleanup(EN_GLOBALCTX *encrGctx)
{
 int  returncode;
 sb_RNGCtx  rngctx;
 void  *sbctx;
 sbctx = encrGctx->en_sbgctx;
 if (encrGctx->en_grngctx)
 {
  (void)sb_ANSIRngDestroy((sb_RNGCtx *)&encrGctx->en_grngctx,
    sbctx);
 }
 if (encrGctx->en_gyieldctx)
 {
  (void) sb_YieldDestroy((sb_YieldCtx *)&encrGctx->en_gyieldctx,
    sbctx);
 }
#if USE_SB4
 if (sbctx)
 {
  (void)sb_SBContextDestroy(&encrGctx->en_sbgctx);
 }
#endif /* USE_SB4 */
}
/*
** EN_GENERATERANDOMDATA
**
** Generate requested size of random data.
**
** Parametersr:
** encrGctx - (in) pointer global SB context structure
** rdbuf    - (out)buffer to hold random data
** rdlen  - (in) number of bytes of requested random data
**
** Returns
** SUCCEED/FAIL
**
*/
int
en_generateRandomData(EN_GLOBALCTX *encrGctx, unsigned char *rdbuf,
  size_t rdlen)
{
 int  returncode;
 void *  sbctx;
 sb_RNGCtx rngctx;
 sbctx = encrGctx->en_sbgctx;
 rngctx = encrGctx->en_grngctx;
 returncode = sb_RngGetBytes(rngctx, rdlen, rdbuf, sbctx);
 if (returncode != SB_SUCCESS)
 {
  TRACESB("sb_RngGetBytes failed with error: %s\n",
   en__geterr(returncode));
  goto fail;
 }
 return SUCCEED;
fail:
 return FAIL;
}
/*
**  EN_SEED
**
**  Callback function to create a seed for random number generation
**
**  Parameters:
```

```
** rsourceParam - (in) Caller supplied data param on RNG function
** bufsize  -      (in) Number of bytes of seed
** buf  -      (out) Seed
** sbctx -         (in) sbContext or memory callback data.
**
**  Returns:
** SB_SUCCESS
** SB_ERR_BAD_INPUT_BUF_LEN;
**
*/
SYB_STATIC int
en_seed(void *rsourceParam, size_t bufsize, BYTE *buf, void *sbctx)
{
 DATE currtime;
 BYTE *ptr;
 int i;
 if (bufsize > SEEDSIZE_BYTES)
 {
  /* ENCRCOLS_RESOLVE: CertiCom doc doesn't specify
  ** that the size will be that of the original
  ** seed, but it shouldn't be greater than.
  */
  SYB_ASSERT(0);
  return SB_ERR_BAD_INPUT_BUF_LEN;
 }
 /* SB_RESOLVE: Figure out a better/more secure algorithm for
 ** generating a seed. For now fill up the buffer with copies
 ** of the date.
 */
 (void)utget(&currtime);
 ptr = (void *) &currtime.dttime;
 for (i = 0; i < bufsize; i += 4)
 {
  MEMMOVE(ptr, buf +i, 4);
 }
 return SB_SUCCESS;
}
/*
**  EN__YIELDCALLBACK
**
**   Callback function for Security Builder to call to yield
**
**  Parameter:
** yieldData (in) Pointer (unused) to ASE data through SB API
**
**  Returns:
** 0
**
*/
int
en_yieldCallback(void *yieldData)
{
 (void) yieldData; /* Reference it to avoid warning */
 upyield();
 return 0;
}
/*
**  EN__GETERR
**
**   Map a Security Builder Error Code to a string version of the same.
**
**  Parameter:
** sb_errcode (in) code to map to string
**
**  Returns:
** Pointer to err string
**
*/
```

```
SYB_STATIC char *
en__geterr(int sb_errcode)
{
 int i;
 i = 0;
 while (Err_lookup[i].ecode != 0 && Err_lookup[i++].ecode != sb_errcode)
 {
  i++;
 }
 return Err_lookup[i].edesc;
}
/*
**  EN_MALLOC
**
**  Callback function to allocate ASE memory for Security Builder needs
**
**  Parameters:
** size    - (in) Caller supplied data param on RNG function
** cbData  - (in) Pointer (unused) to ASE data through SB API
**
**  Returns:
** Nothing
**
*/
void *
en_malloc(size_t size, void * cbData)
{
 return ubfalloc(Kernel->kencr_mempool, size);
}
/*
**  EN_FREE
**
**  Callback function to free ASE memory used by Security Builder
**
**  Parameters:
** ptr     -  (in) Pointer to memory to be freed
** cbData  - (in) Pointer (unused) to ASE data through SB API
**
**  Returns:
** Nothing
**
*/
void
en_free(void *ptr, void *cbData)
{
 ubffree(Kernel->kencr_mempool, ptr);
 return;
}
/*
**  EN_MEMCPY
**
**  Callback function to copy ASE memory used by Security Builder
**
**  Parameters:
** dst     -  (in) Pointer to destination memory
** src     -  (in) Pointer to source memory
** len     -  (in) Number of bytes to be copied
** cbData  - (in) Pointer (unused) to ASE data through SB API
**
**  Returns:
** Nothing
**
*/
void
en_memcpy (void *dst, const void *src, size_t len, void *cbData)
{
 MEMMOVE(src, dst, len);
 return;
```

```
}
/*
**  EN_MEMCMP
**
**  Callback function to copy ASE memory used by Security Builder
**
**  Parameters:
** block1  - (in) Pointer to memory to be copied
** block2  - (in) Pointer to destination of copy operation
** len     -  (in) Number of bytes to be copied
** cbData  - (in) Pointer (unused) to ASE data through SB API
**
**  Returns:
** Nothing
**
*/
int
en_memcmp(const void *block1, const void *block2, size_t len, void *cbDa
ta)
{
 return memcmp(block1, block2, len);
}
/*
**  EN_MEMSET
**
**  Callback function to initialize ASE memory for Security Builder's ne
eds
**
**  Parameters:
** buf    - (in) Pointer to memory to be initialized
** len    - (in) Number of bytes to be initialized
** cbData - (in) Pointer (unused) to ASE data through SB API
**
**  Returns:
** Nothing
**
*/
void
en_memset(void *buf, int value, size_t len, void *cbData)
{
 memset(buf, value, len);
 return;
}
/*
** EN_CONCATBYTES
**
** Purpose:
** Mix and concatenate up to three byte strings and produce
** a digest of the result, suitable for a key.  This function is
** obscurely named on purpose.
** The algorithm is as follows:
**  SHA1(mix1(mix2(STR1, reverse(STR2)), xor(STR3)))
** where
**     - reverse() reverses the bytes of a string
**     - xor() performs exclusive-or on the current and next byte
**  in a string.  Last byte is unchanged.
**     - mix1() takes byte 1 from 1st arg, byte 1 from 2nd arg;
**   then byte 2 from 1st arg, byte 2 from 2nd arg, etc.
**  up to length of smaller string.  The rest of the
**  longer string is concatenated to end of mix.
**     - mix2() concats bytes 1 and 2 from 1st arg with byte 1 from
**  2nd arg; then bytes 3 and 4 from 1st arg with byte 2 from
**  2nd arg, until either argument is exhausted.  Remaining
**  bytes from either argument are concatenated to the end
**  of the mix.
**
** Parameters:
** bytes1, bytes2, bytes3 (In) Byte strings
```

```
** len1, len2, len3 (In) Respective lengths
** resbytes  (In) Address of buffer for result
** reslen   (In/Out) Array size/number of bytes filled
**
** Returns:
** SUCCEED/FAIL
**
*/
int
en_concatbytes(EN_GLOBALCTX *encrGctx,
       BYTE *bytes1, int len1,
       BYTE *bytes2, int len2,
       BYTE *bytes3, int len3,
       BYTE *resbytes, size_t *reslen)
{
 BYTE  tmp1buf[EN_INTRNL_KPARTS_LEN];
 BYTE  tmp2buf[EN_INTRNL_KPARTS_LEN*2];
 BYTE tmp3buf[EN_INTRNL_KPARTS_LEN*3];
 BYTE *p1;
 BYTE *p2;
 BYTE *mixp;
 int i, j, k;
 size_t  minlen, maxlen;
 char *errp;
#if STATIC_KEY_DEBUG
 char outbuf[500];
 char *outbp;
#endif /* STATIC_KEY_DEBUG */
 SYB_ASSERT(len1 <= EN_INTRNL_KPARTS_LEN);
 SYB_ASSERT(len2 <= EN_INTRNL_KPARTS_LEN);
 SYB_ASSERT(len3 <= EN_INTRNL_KPARTS_LEN);
 SYB_ASSERT(len1 || len2 || len3);
 SYB_ASSERT(*reslen >= EN_AES_DIGEST_LEN);
 /* Initialize */
 memset(tmp1buf, 0, EN_INTRNL_KPARTS_LEN);
 memset(tmp2buf, 0, EN_INTRNL_KPARTS_LEN*2);
 memset(tmp3buf, 0, EN_INTRNL_KPARTS_LEN*3);
#if STATIC_KEY_DEBUG
 if (len1 > 0)
 {
  outbp = &outbuf[0];
  en_mkhexstr(&outbp, bytes1, len1);
  TRACESB("\nBytes1 input\n");
  TRACESB(&outbuf[0]);
 }
 if (len2 > 0)
 {
  outbp = &outbuf[0];
  en_mkhexstr(&outbp, bytes2, len2);
  TRACESB("\nBytes2 input\n");
  TRACESB(&outbuf[0]);
 }
 if (len3 > 0)
 {
  outbp = &outbuf[0];
  en_mkhexstr(&outbp, bytes3, len3);
  TRACESB("\nBytes3 input\n");
  TRACESB(&outbuf[0]);
 }
#endif /* STATIC_KEY_DEBUG */
 if (len2 > 0)
 {
  /* Reverse second byte string */
  for (i = len2-1, j = 0; i >= 0; i--, j++)
  {
   tmp1buf[j] = *(bytes2+i);
  }
#if STATIC_KEY_DEBUG
```

```
  outbp = &outbuf[0];
  en_mkhexstr(&outbp, &tmp1buf[0], j);
  TRACESB("\nSecond buffer reversed\n");
  TRACESB(&outbuf[0]);
#endif /* STATIC_KEY_DEBUG */
 }
 if (len1 > 0 && len2 > 0)
 {
  /*
  ** Using shorter len, mix bytes1 and bytes2 buffers, taking
  ** one byte from each in turn
  */
  minlen = (len1 > len2) ? len2 : len1;
  maxlen = (len1 > len2) ? len1 : len2;
  j = 0;
  for (i = 0; i < minlen; i++)
  {
   tmp2buf[j++] = bytes1[i];
   tmp2buf[j++] = tmp1buf[i];
  }
  /* Append remainder of longer string */
  p1 = (len1 > minlen) ? bytes1 : tmp1buf;
  for (i = minlen; i < maxlen; i++)
  {
   tmp2buf[j++] = *(p1+i);
  }
  /* Save for concatenation with bytes3 */
  mixp = &tmp2buf[0];
  /* Done with tmp1buf */
  memset(tmp1buf, 0, EN_INTRNL_KPARTS_LEN);
#if STATIC_KEY_DEBUG
  outbp = &outbuf[0];
  en_mkhexstr(&outbp, mixp, j);
  TRACESB("\nMixture of 1st and 2nd components\n");
  TRACESB(&outbuf[0]);
#endif /* STATIC_KEY_DEBUG */
 }
 else
 {
  mixp = (len1 > 0) ?
   bytes1 : (len2 > 0) ? &tmp1buf[0] : NULL;
 }
 if (len3 > 0)
 {
  /* Use whichever temporary buffer is available */
  p2 = (len1 > 0 && len2 > 0) ? &tmp1buf[0] : &tmp2buf[0];
  p1 = bytes3;
  /* XOR each byte of bytes3 with the next consecutive byte */
  for (i = 0; i < len3-2; i++)
  {
   *(p2+i) = *(p1+i);
   *(p2+i) ^= *(p1+i+1);
  }
  *(p2+len3-1) = *(p1+len3-1);
#if STATIC_KEY_DEBUG
   outbp = &outbuf[0];
   en_mkhexstr(&outbp, p2, len3);
   TRACESB("\nXOR of third component\n");
   TRACESB(&outbuf[0]);
#endif /* STATIC_KEY_DEBUG */
  /*
  ** Now put buf3 into the mix, 1 byte for every 2 bytes of
  ** already mixed bytes
  */
  if (mixp)
  {
   i = j = k = 0;
   while (i < len1+len2-1 && k < len3)
```

```
    {
     tmp3buf[j++] = *(mixp+(i++));
     tmp3buf[j++] = *(mixp+(i++));
     tmp3buf[j++] = *(p2+(k++));
    }
    while (i < len1+len2)
    {
     tmp3buf[j++] = *(mixp+(i++));
    }
    while (k < len3)
    {
     tmp3buf[j++] = *(p2+(k++));
    }
    mixp = &tmp3buf[0];
#if STATIC_KEY_DEBUG
    outbp = &outbuf[0];
    en_mkhexstr(&outbp, mixp, len1+len2+len3);
    TRACESB("\nMixture of third component\n");
    TRACESB(&outbuf[0]);
    TRACESB("\n");
#endif /* STATIC_KEY_DEBUG */
   }
   else
   {
    mixp = p2;
   }
  }
  /* Now digest the mixed byte strings */
  if (!en_sha1_digest(encrGctx, mixp, (size_t)len1+len2+len3,
   resbytes, reslen, &errp))
  {
   return FAIL;
  }
  return SUCCEED;
}
#if STATIC_KEY_DEBUG
SYB_STATIC void
en_mkhexstr(char **outbp, BYTE *datap, int datlen)
{
 int i;
 sprintf(*outbp, "0x");
 *outbp += 2;
 for (i = 0; i < datlen; i++)
 {
  sprintf(*outbp, "%02x", *datap++);
  *outbp += 2;
 }
}
#endif /* STATIC_KEY_DEBUG */
#else  /* USE_SECURITYBUILDER */
/*
** Function stubs to satisfy compiler on platforms where
** Security Builder libraries are not supported.
*/
int
en_sha1_digest(EN_GLOBALCTX * encrGctx, BYTE *message, size_t msglen,
  BYTE *digest, size_t *diglen, char **errdesc)
{
 SYB_ASSERT(0);
 return FAIL;
}
int
en_aes_createsymkey(EN_GLOBALCTX *encrGctx, size_t keysize, int status,
 BYTE *keybuf, size_t *kbuflen, char **errdesc)
{
 SYB_ASSERT(0);
 return FAIL;
}
```

```
int
en_aes_beginCryptOper(EN_GLOBALCTX *encrGctx, EN_LOCALCTX *encrLctx,
  unsigned char *key, size_t keysize, int status,
  unsigned char *ivbuf, int ivlen, char **errdesc)
{
 SYB_ASSERT(0);
 return FAIL;
}
int
en_aes_encrypt(EN_GLOBALCTX *encrGctx, EN_LOCALCTX *encrLctx,
  unsigned char *plaintext, size_t plaintlen,
  unsigned char *ciphertext, char **errdesc)
{
 SYB_ASSERT(0);
 return FAIL;
}
int
en_aes_decrypt(EN_GLOBALCTX *encrGctx, EN_LOCALCTX *encrLctx,
  unsigned char *ciphertext, size_t ciphtlen,
  unsigned char *plaintext, char **errdesc)
{
 SYB_ASSERT(0);
 return FAIL;
}
int
en_aes_endCryptOper(EN_GLOBALCTX *encrGctx, EN_LOCALCTX *encrLctx,
  char **errdesc)
{
 SYB_ASSERT(0);
 return FAIL;
}
int
en_generateRandomData(EN_GLOBALCTX *encrGctx, unsigned char *rdbuf,
  size_t rdlen)
{
 SYB_ASSERT(0);
 return FAIL;
}
#endif /* USE_SECURITYBUILDER */
```